Software Engineering with Ada

Student Notes

Executive Overview

- **Motivation, History, Strategy**
  - Software Crisis
  - Software for Embedded Computer Systems - 1974
  - Components of the Implementation of the Strategy
  - Why a New Language?
  - Three Legs of the Language
  - Ada continues the tradition
- Themes & Examples
  - Effective use of Ada
  - Software Engineering Principles and Ada
  - Object-Oriented Design and Ada
  - Alternative Solutions to Problems and their Impact on Software Goals
- Emerging Software Scene
  - Technology
  - Human Resources
  - Business Practices
  - Applications

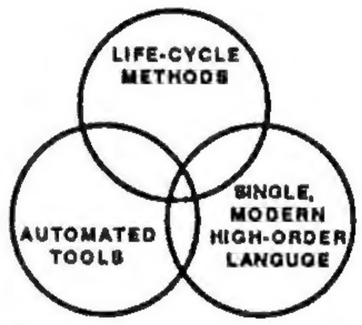
Software Crisis

- Software late, over cost, unreliable, difficult to maintain
- Skyrocketing software expenditures
- Projections of manpower falling behind
- Symptoms were most severe in embedded systems

Software for Embedded Computer Systems - 1974

- More than half of DoD Software Expenditures
- The Facts
  - Unique hardware with unique assembly language for each weapon system
  - Several hundred such languages
  - Everything special purpose and thus single use (software, training, experience)
  - No cost spreading through multiple use
- The Results
  - High life cycle cost in time and money for both development and modification
  - Low quality (Reliability, Efficiency, Modifiability)
- The Strategy
  - Lifecycle Engineering approach
  - Multiple use of software, training, experience
  - Automation of much of the process

Components of the Implementation of the Strategy



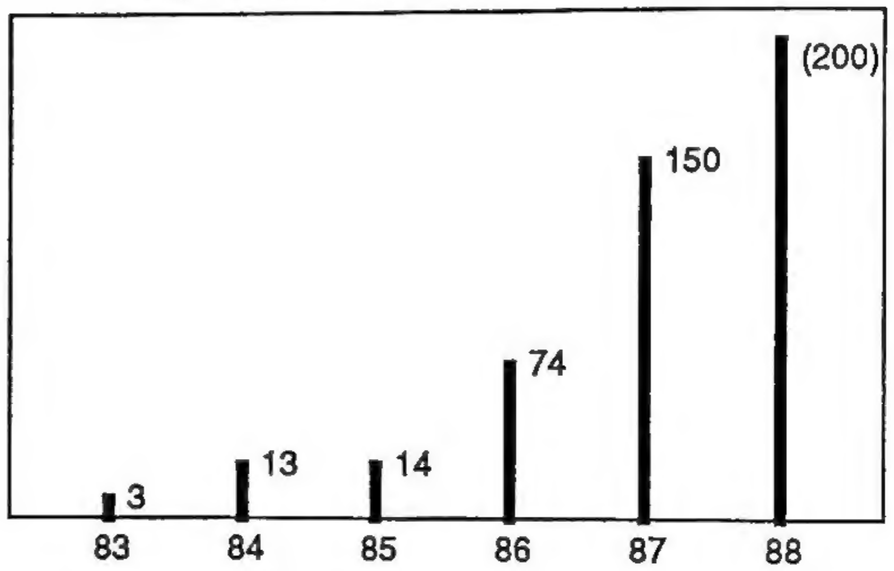
- Life-Cycle Methods
  - Recognize software as a large, complex, long-lived creation to be manipulated and used by many
  - Coordinate large numbers of people over long periods of time
  - Improve maintainability, readability etc.
- Automated Tools
  - Recognize that many methods involve tedium and intricacy
  - Make methods cost effective through automation
- Single, Modern, High-order Language
  - Single: multiple use of tools, people, software, etc.
  - Modern: Permits expression and enforcement of encapsulation, reuse, concurrency, real-time, etc.

Why a New Language?

- No existing language adequately addressed the requirements
- Result is a highly sophisticated tool whose mastery requires considerable training and experience
- A natural extension of the evolutionary chain of programming languages

Three Legs of the language

- Standard Definition
  - Ansi/Mil Std 1815a (1983)
  - ISO Std (1987)
- Validation
  - Approximately 3000 test programs
  - Assures compliance with standard
  - Annual revalidation required
- Many Validated Implementations



Ada continues the tradition of providing facilities to describe objects at ever higher levels of abstraction

Problem Space -- Very High Level Application Specific Problem Oriented Languages

1980	Ada	(Packages, Generics, Tasking, Strong Typing, Extensibility)
1973	ALPHARD, EL1, CLU	(experimental Abstraction and Tasking Facilities)
1969	Pascal	(Data Structures)
1960	Algol	(Formal Definition, Block Structure, Control Structures, Parameter Mechanisms)
1954	Fortran	(Algebraic Expressions, Parameterized Procedures)
1951	IBM 650	Assembly Language (Locations, Mnemonics)
194X	Machine Language	(All work done by programmer)

Machine Space -- Low Level Hardware Specific Machine Languages

MACHINE LANGUAGE

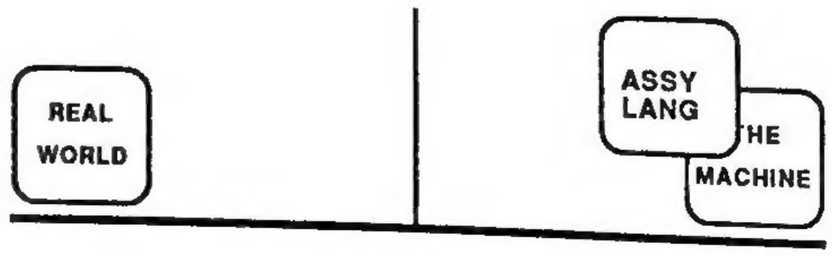
NO ABSTRACTION

```
01100011
11001110
00001100
...
```

ASSEMBLY LANGUAGE

ABSTRACTION OF LOCATION

```
      BRZ   L1
      LDA   Y
      ADD   Z
L1    ...
```



FORTRAN (1954)

ABSTRACTION OF EXPRESSIONS

X = (Y + Z) * V

INSTEAD OF

```
LDA  Y
ADD  Z
MLT  V
STA  X
```

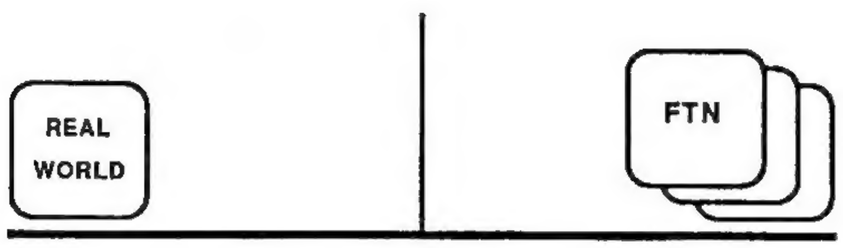

ALGOL (1960)

ABSTRACTION OF CONTROL

if-then-else, while, repeat, etc

INSTEAD OF

```
L1:  ---
     ---
     GO TO L2
L3:  ---
     ---
     GO TO L1
L2:  ---
```

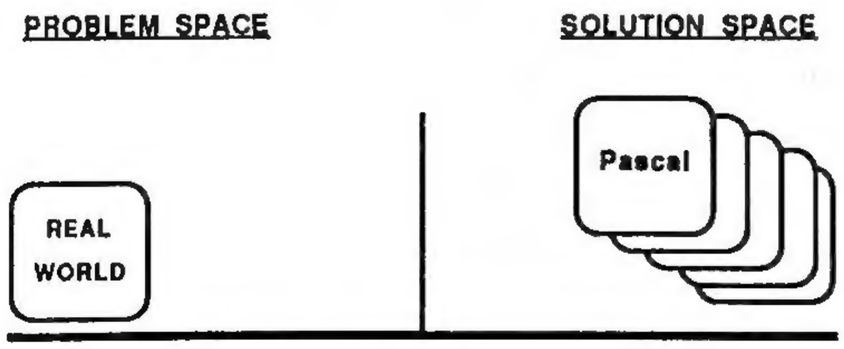
Pascal (1970)

ABSTRACTION OF DATA

**Arrays, records, sets,
arrays of records of arrays,
enumerated values (SUN, MON, ..., SAT)**

INSTEAD OF

**Low-level data structures,
Great reliance on the integers**



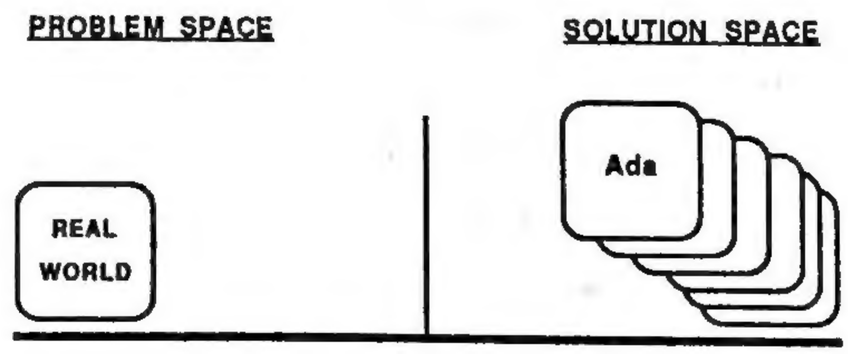
Ada (1980)

ENFORCED ABSTRACTIONS

**ENCAPSULATION/LOCALIZATION
PROCEDURAL ABSTRACTION
INFORMATION HIDING
ABSTRACT DATA TYPES**

INSTEAD OF

**Reliance on standards ("Thou shalt not . . .")
to enforce good software engineering practices**



Executive Overview

- Motivation, History, Strategy
  - Software Crisis
  - Software for Embedded Computer Systems - 1974
  - Components of the Implementation of the Strategy
  - Why a New Language?
  - Three Legs of the Language
  - Ada continues the tradition
- → **Themes & Examples**
  - Effective use of Ada
  - Software Engineering Principles and Ada
  - Object-Oriented Design and Ada
  - Alternative Solutions to Problems and their Impact on Software Goals
- Emerging Software Scene
  - Technology
  - Human Resources
  - Business Practices
  - Applications

Themes

- Effective use of Ada's greater expressive power
  - to express solutions in problem space terms
  - to express information about the software itself
  - to express more precise information about the computation itself
  - Information is expressed in compilable Ada Code processable by the compiler and other tools
- Several ways to approach the use of Ada's expressive power
  - Software Engineering Principles and Ada
  - Object-Oriented Design and Ada
  - Alternative Solutions to Problems and their Impact on Software Goals

Effective Use of Ada

- Effective Use of Ada yields many benefits
  - Problem space fidelity and direct expressibility
  - Explicit expression of design decisions
  - Enforced information hiding
  - Isolation of machine and system dependencies
  - Precise control over values and value checking
  - Clean and understandable error handling
  - Increased automatic control (and reduced manual control) of the software
- Features Key to the Effective Use of Ada
  - User-defined Data Types
  - Packaging
  - Separate Compilation
  - Exception Handling
  - Generics
  - Tasking

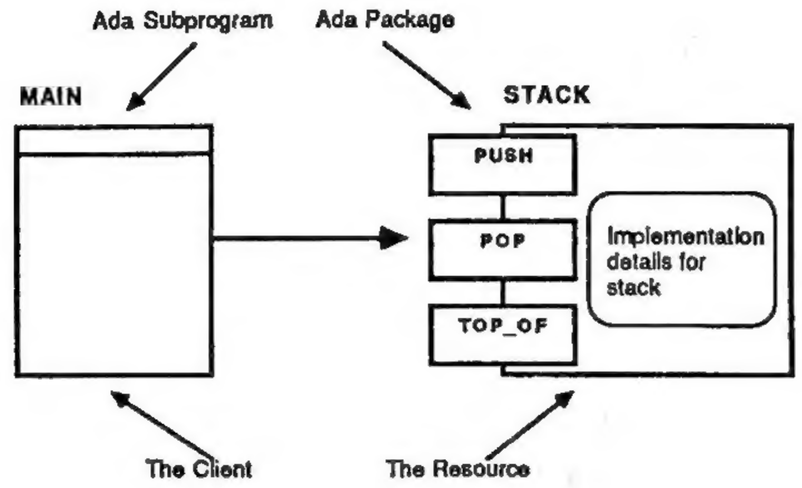
User-defined Data types

- Ada is a strongly-typed language
- The language will enforce user-defined restrictions on data

```
type WORK_AGE is range 18 .. 65;

type VOLTAGE is delta 0.25 range 100.0 .. 500.0;

type SPEED is range 0 .. 3000;

subtype AUTO_SPEED is SPEED range 0 .. 250;

subtype LEGAL_SPEED is SPEED range 0 .. 65;

type AIRCRAFT is (FRIEND, FOE, UNKNOWN);

type GENDER_TYPE is (MALE, FEMALE);

type PERSONNEL_RECORD is
    record
        NAME : STRING ( 1 .. 30);
        AGE : WORK_AGE;
        GENDER : GENDER_TYPE;
    end record;
```

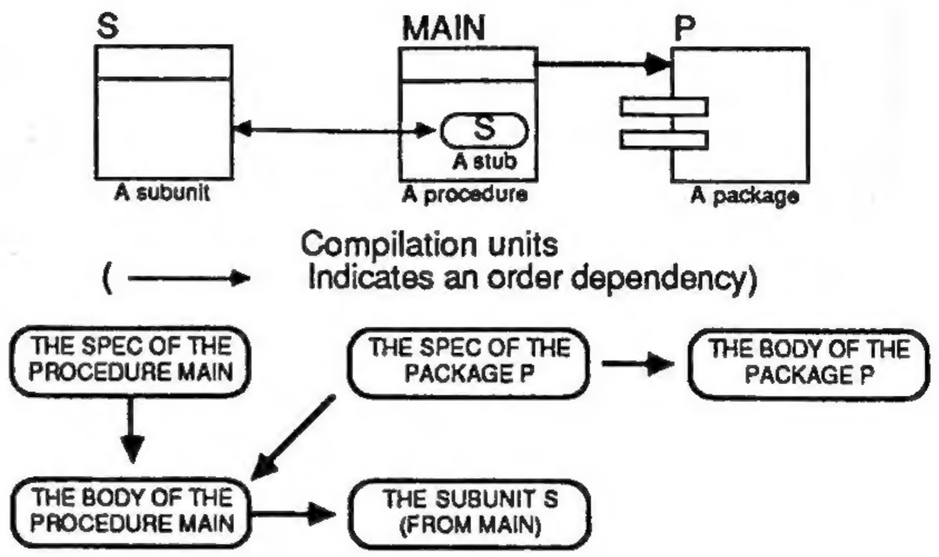
Packaging

- An encapsulation mechanism
- Allows client (user) to focus on the functionality of a resource without worrying about its actual implementation



Separate Compilation

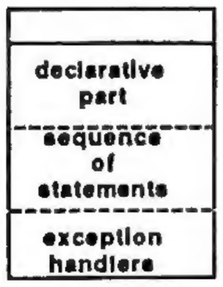
- The library (an integral part of the language) contains compilation units
- Compilation units can be submitted for compilation separately and the library will maintain a history of information
- Compilation units form a partial ordering within the library



Exception Handling

- An exception is a signal that something has gone wrong (divide by zero, out-of-range value, etc.)
- An Exception handler is a portion of code that is executed when an error occurs within the associated sequence of statements
- Exceptions not handled are 'propagated' outward

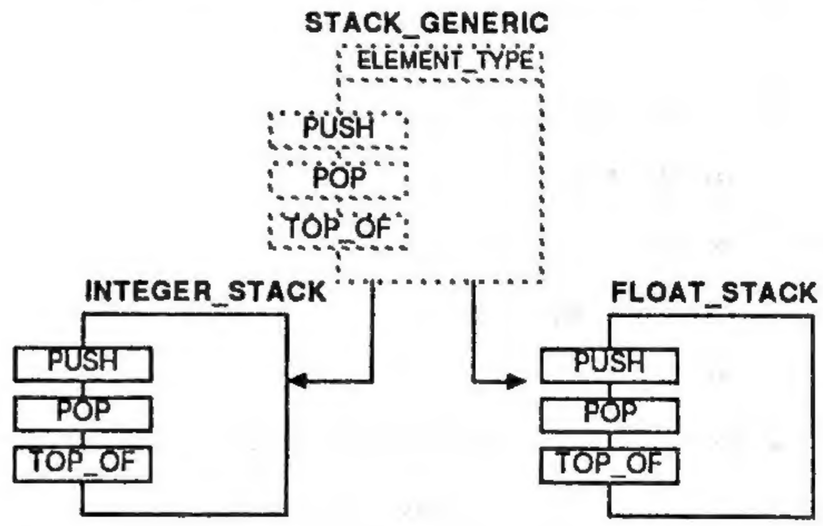
An Ada Subprogram




```
procedure SAMPLE is
  I,J,K : INTEGER := 0;
begin
  . . .
  J := 17;
  I := J/K;
  . . .
exception
  when NUMERIC_ERROR =>
  . . .
end SAMPLE;
```


Generics

- A high-order language 'macro'
- Allows similar subprograms and packages to be created from a template (generic unit)



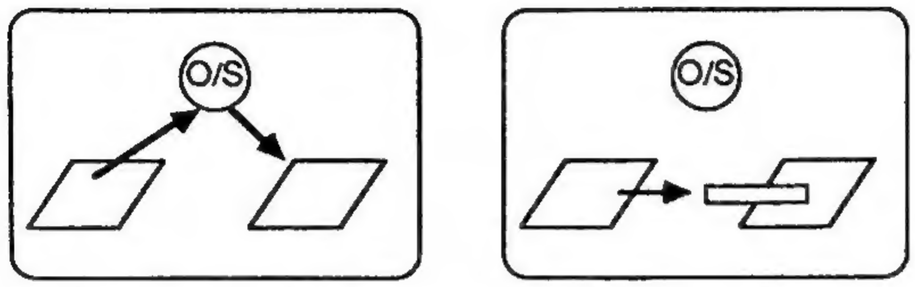
Generic Instantiation

```
package INTEGER_STACK is new STACK_GENERIC (ELEMENT_TYPE => INTEGER);
package FLOAT_STACK is new STACK_GENERIC (ELEMENT_TYPE => FLOAT);
```

Tasking

- Ada provides a model of concurrency which is completely defined in the high-order language
- Reliance on operating system resources is not required

CONCURRENCY



Traditional approach

Ada approach

SOFTWARE ENGINEERING GOALS

- MODIFIABILITY
  - -- Controlled change
  - -- Logical invariance to physical change
  - -- Solution space maps the problem space
- EFFICIENCY
  - -- Time/space tradeoff
  - -- Microefficiency often considered too early
  - -- Macroefficiency achieved by unified understanding of the problem
- RELIABILITY
  - -- Prevention of failure
  - -- Recovery from failure
  - -- Often considered too late
- UNDERSTANDABILITY
  - -- Many different views to deal with
  - -- 'Golden rule' of software applies
  - -- Code is written once but is read far more often than that

Software Engineering Principles and Ada

Ada allows decisions based on Software Engineering Principles to be explicitly reflected in compilable code, permitting automatic checking.

- Software Engineering Principles
  - •• Abstraction
  - •• Information Hiding
  - •• Encapsulation
  - •• Modularity
- Features key to reflecting these principles
  - •• Ada's program units (subprograms, packages, tasks and generics) help implement these principles
  - •• Ada's scope and visibility rules help enforce these principles

Abstraction

The process of identifying the important properties of the phenomenon being modeled and ignoring (for the moment) the underlying details.

- Each level of decomposition represents an abstraction
- Each level must be completely understood as a unit
- Abstraction applies to data as well as to algorithms
- Facilitates mapping from problem space to solution space

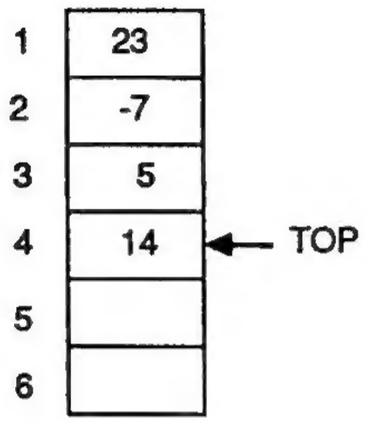
Information Hiding

- Make details of an implementation inaccessible
- Enforce defined interfaces
- Focus on the abstraction of an object by suppressing the underlying details
- Prevent high-level decisions from being based on low-level characteristics

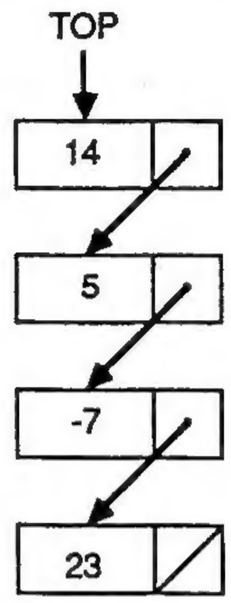
Abstraction and Information Hiding

A STACK is an abstract object with abstract operations PUSH and POP (among others). The user of a stack ought not be concerned about how the object (or the operations) are implemented.

IMPLEMENTATION A

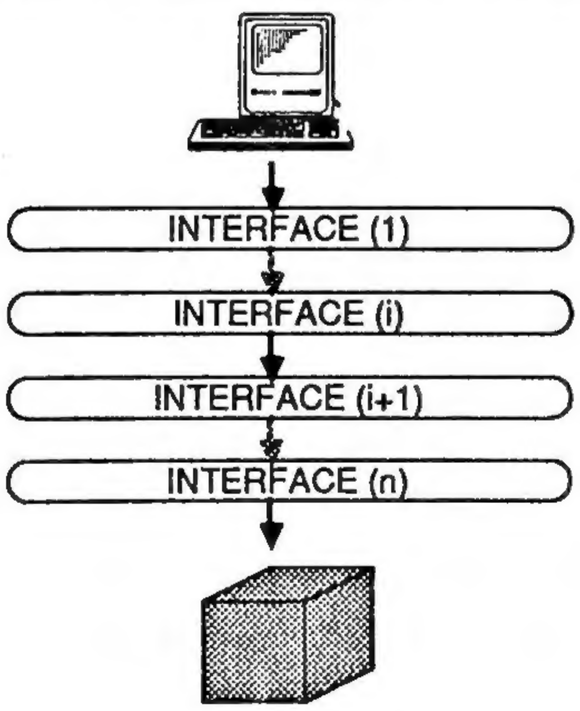


IMPLEMENTATION B



System Interfaces

- A well engineered system will most often consist of a collection of layered interfaces



Software Interfaces

- Outside View
- Abstract View
- Functional View
- Client View
- Schema

The outside view provides the abstraction of the interface and does not concern itself with how the features of the interface are actually implemented

INTERFACE

- Inside View
- Implementation
- Detailed View

Information on this side of the interface is hidden from the client. The client must rely only on the information contained in the outside view

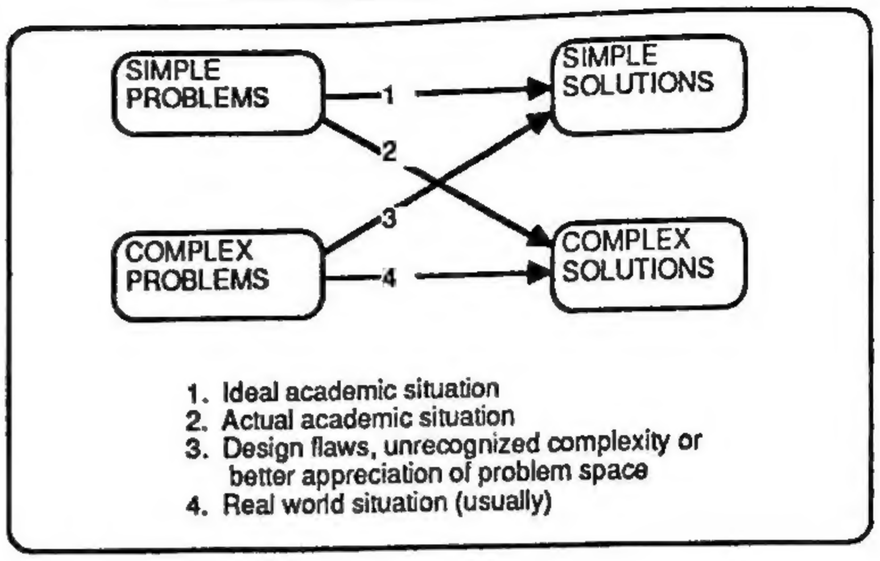
INTERFACE

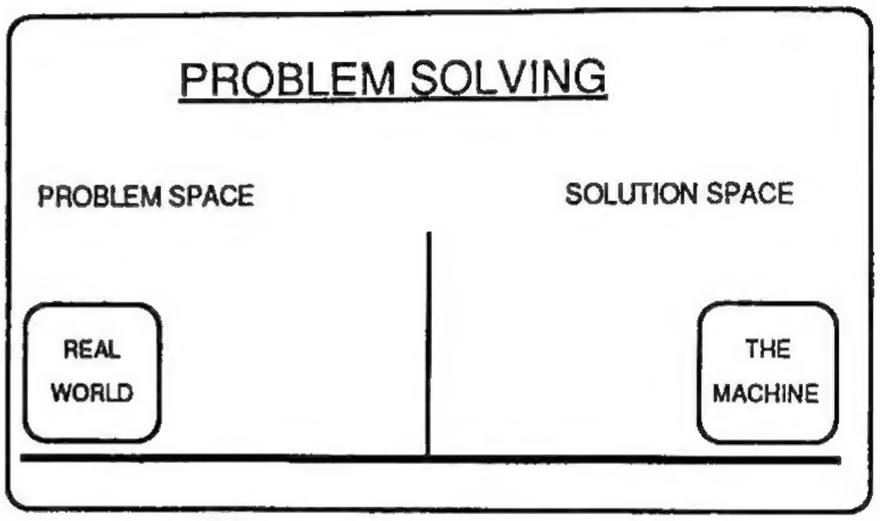
Notice that the implementor (of the inside view) of one interface is likely the client (with outside view) of another interface.

Ada Interfaces

- All Ada program units (subprograms, packages, tasks and generics) are composed of two parts
  - The specification is the outside view and provides the abstraction of the resource
  - The body is the inside view and provides the implementation of the resource
- The client of the resource sees only the specification. The client can never see "inside" the body of the resource
- Therefore, the body can change radically and, as long as the specification is still implemented, the client is unaffected by the change

Object-Oriented Design





Object-Oriented Design and Ada

Ada permits a near-verbatim implementation of an Object-oriented Design, permitting automatic checking.

- Object-oriented Design
  - Objects
  - Operations
  - Interface
  - Errors in operations
- Features key to implementing an object-oriented design
  - Packages and Generics implement objects
  - Subprograms implement operations
  - Exceptions map problem-space errors discovered while executing operations

Object-Oriented Design

A means of mapping problem-space 'objects' onto solution-space constructs

An Object

- Has state
- Is characterized by its operations
  - Constructors - change state
  - Selectors - report state
- Has restricted visibility of and by other objects
- Can be viewed in two ways
  - By its specification (outside, abstract view)
  - By its implementation (inside, detailed view)
- Is a distinct (perhaps unique) instance of some class

Object-Oriented Design

Identify the objects — Overhead Projector

Identify the operations

Constructors

- Turn_On
- Turn_Off
- Change_Bulb
- Plug_In
- Focus

Selectors

- Projector_is_on
- Bulb_is_burnt_out
- Is_plugged_in
- Weight

Establish Interface (Outside view)

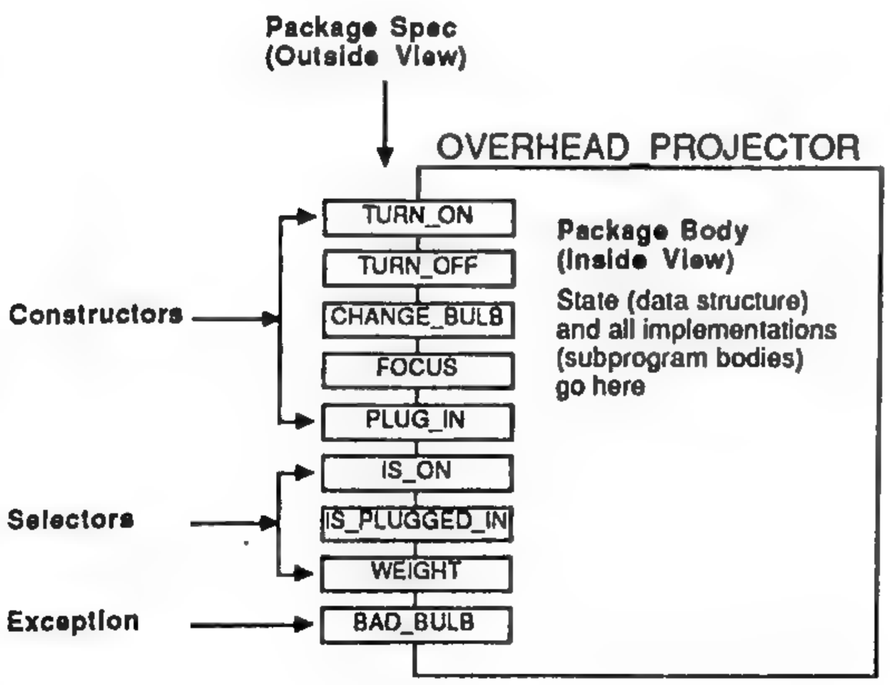
Implement the object (Inside view)

- Decide on implementation of state
- Implement each operation

Object-Orientation and Ada

Object-Orientation	Ada Construct
Object	Package or generic package
Outside View	Package Specification
Inside View	Package body (and private part)
Constructor	Procedure (Usually)
Selector	Function (Usually)
Errors in Operations	Exception
Object Class	Package with private type
Abstract Object	Package
Names of Objects	Variables
State (object class)	In instance of private type
State (Abstract Object)	In package body

An Example of an Ada Object



An Example of an Ada Object

```
with BULB_DATA, MEASURES;
package OVERHEAD_PROJECTOR is
   procedure TURN_ON;
   procedure TURN_OFF;
   procedure CHANGE_BULB (B : in BULB_DATA.BULB);
   procedure FOCUS;
   procedure PLUG_IN;

   function IS_ON return BOOLEAN;
   function IS_PLUGGED_IN return BOOLEAN;
   function WEIGHT return MEASURES.WEIGHT_TYPE;

   BAD_BULB : exception;

end OVERHEAD_PROJECTOR;

-------------------------------------------------------------

package body OVERHEAD_PROJECTOR is

   type PROJECTOR_TYPE is . . .
→  THE_PROJECTOR : PROJECTOR_TYPE;
   . . .
   procedure TURN_ON is
   begin
      . . .
   end TURN_ON;
   . . .
   . . .
end OVERHEAD_PROJECTOR;
```

Alternative Solutions to Problems and their impact on Software Goals

- Ada offers many solutions to any problem, and selection of which solution to use is frequently determined by high-level goals. Understanding trade-offs is thus key
- Software goals
  - Performance
  - Portability
  - Reuse and Reusability
  - Testability
  - Maintainability
  - Reliability
  - Problem Domain Fidelity
  - Robustness
  - Recompilation Efficiency, Etc.
- Features key to goal achievement
  - Type selection
  - Tasking implementation
  - Generics
  - Reliance on data structures vs statements
  - Separate compilation, Etc

Executive Overview

- Motivation, History, Strategy
  - Software Crisis
  - Software for Embedded Computer Systems - 1974
  - Components of the Implementation of the Strategy
  - Why a New Language?
  - Three Legs to the Language
  - Ada continues the tradition
- Themes & Examples
  - Effective use of Ada
  - Software Engineering Principles and Ada
  - Object-Oriented Design and Ada
  - Alternative Solutions to Problems and their Impact on Software Goals
- → **Emerging Software Scene**
  - Technology
  - Human Resources
  - Business Practices
  - Applications

Technology

- Hardware
- Life-cycle Methodologies
- Software Tools and Environments
- Reuse Technology

Human Resources

- Shortage of Qualified Ada personnel
- Professional Standards
- Training
- Experience

Business Practices

- Closer Control and visibility are possible
- New methods need tolerance and encouragement
- Reuse technology will increase build/buy options
- Dod Policy

DODD 3405.1 "Computer Programming Language Policy"

Signed 2 APR 1987

(1) Ada shall be the single, common, computer programming language for Defense computer resources used in intelligence systems, for the command and control of military forces, or as an integral part of a weapon system.

(2) Programming languages other than Ada that were authorized and being used in full-scale development may continue to be used through deployment and for software maintenance, but not for major software upgrades.

(3) Ada shall be used for all other applications, except when the use of another approved higher order language is more cost-effective over the application's life-cycle.

(4) DoD-Aproved Higher Order Programming Languages

- Ada
- C/ATLAS
- COBOL
- CMS-2M
- CMS-27
- FORTRAN
- JOVIAL(J73)
- Minimal BASIC
- Pascal
- SPL/1

DODD 3405.2 "Use of Ada in Weapon Systems"

Signed 30 MAR 1987

(1) Ada shall be the single, common, high-order programming language, effective immediately;

(2) use of validated Ada compilers is required; and

(3) an Ada-based program design language (PDL) shall be used during the designing of software. Use of a PDL that can be successfully compiled by a validated Ada compiler is encouraged in order to facilitate the portability of the design.

Applications

- Ada in Europe
- Non-DoD Ada Experience
- Real-Time
- MIS

Ada In Europe

- Used in all NATO military systems as/of January 1, 1986
- Several validated Ada compilers
- Compiler implementations by UK, Denmark, France, West Germany, Finland, USSR
- Ada adopted as an ISO standard (12 Mar 87)
- Denmark and Spain jointly writing queuing software (first European commercial venture)
- Denmark and France jointly writing FAA S/W
- UK adopts Ada in favor of CORAL
- Germany accepts only Ada and PEARL for embedded systems
- Sweden mandates Ada for Real-time systems effective January 1987
- Used for two major Finnish banking systems (2M LOC)
- Many Ada textbooks written by Europeans
- Joint Sweden, Denmark, Finland navy project

Non-DoD Ada Experience

- CBT system (McDonnell-Douglas)
- Business Software (Intellimac)
- Communications (Singer-Librascope)
- Industrial Process Control (MOOG)
- Artificial Intelligence (Intellimac)
- NASA commitment -- manned space station
- CCA -- Distributed Relational Database
- Oil industry -- geophysical software
- FAA

Ada Information Clearinghouse

- GENERAL INFORMATION SERVICES
  - -- On-line Ada-Information directory
  - -- Staff available for phone queries
  - -- Information mailings
- AdaIC NEWSLETTER
- CATALOG OF RESOURCES FOR EDUCATION IN ADA AND SOFTWARE ENGINEERING (CREASE)
- ADAIC INFORMATION
  - -- Ada Bibliography
  - -- Documents Reference List
  - -- Validated Compiler List
  - -- Ada Implementations List
  - -- Classes and Seminars
  - -- Conferences and Programs
  - -- Textbooks
  - -- Calendar of Ada Events

Ada Information Clearinghouse
4550 Forbes Blvd., Suite 300
Lanham MD 20709
(301) 731-8894
(703) 685-1477

Technical Overview

- Ada's Requirements and Design
- Ada From the Top Down
  - •• Subprograms
  - •• Tasks
  - •• Packages
  - •• Generics
  - •• Separate Compilation
- Ada From the Bottom Up
  - •• Character Set
  - •• Reserved Words
  - •• Types
  - •• Statements
  - •• Representation Specifications

Ada DESIGN GOALS

- RECOGNITION OF THE IMPORTANCE OF PROGRAM RELIABILITY AND MAINTAINABILITY
- CONCERN FOR PROGRAMMING AS A HUMAN ACTIVITY
- EFFICIENCY

"We must recognize the strong and undeniable influence that our language exerts on our way of thinking and in fact defines and delimits the abstract space in which we can formulate - give form to - our thoughts."

– Nicklaus Wirth, 1974

STEELMAN REQUIREMENTS

- STRUCTURED CONSTRUCTS
- STRONG TYPING
- RELATIVE AND ABSOLUTE PRECISION
- INFORMATION HIDING AND DATA ABSTRACTION
- CONCURRENT PROCESSING
- EXCEPTION HANDLING
- GENERIC DEFINITION
- MACHINE DEPENDENT FEATURES

What kind of language is Ada?

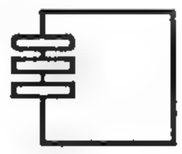
- an algorithmic language
  - -- subprograms (functions and procedures)
  - -- structured control statements
  - -- complete data structuring capability
- a design language
  - -- packages, tasks, subprograms for decomposition
  - -- separate compilation for top-down design
  - -- library units for bottom-up design
  - -- generic units for reuseability
- a systems programming language
  - -- tasking for concurrent processes
  - -- representation specs for 'bit twidling'
  - -- exception handling
  - -- hardware interrupt recognition
- an extendable language
  - -- can be tallored to a given application area

PROGRAM UNITS (Ada from the top)

- SUBPROGRAMS
  - -- Functions and Procedures
  - -- Main program
  - -- Abstract operations

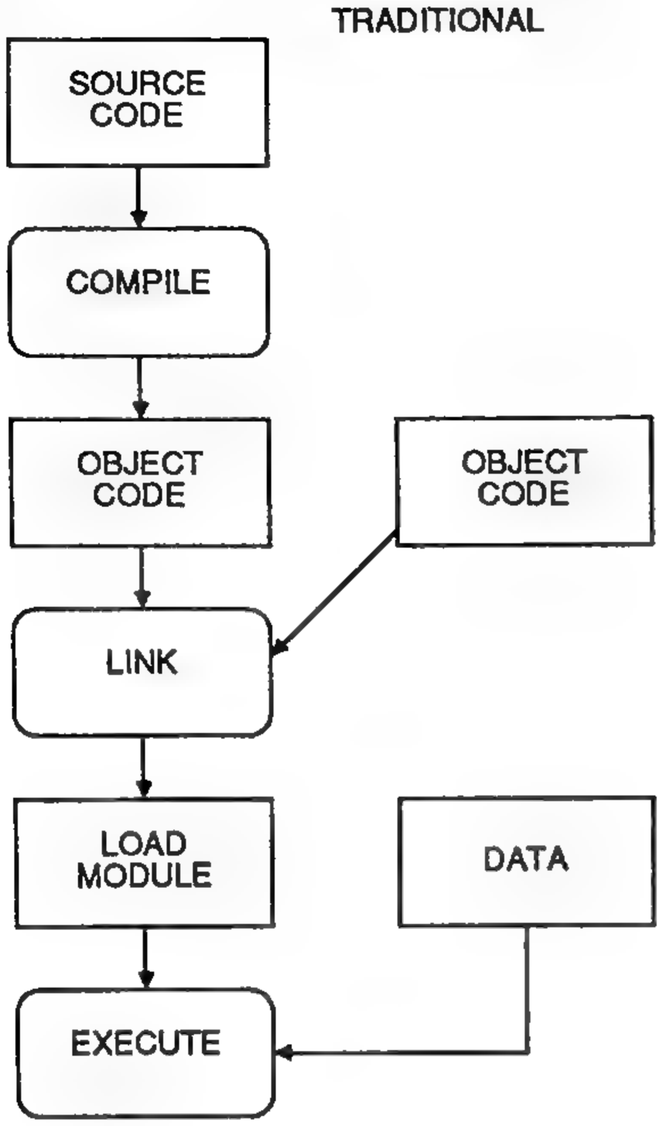
- TASKS
  - -- Parallel Processing
  - -- Real-Time
  - -- Interrupt Handling

- PACKAGES
  - -- Encapsulation
  - -- Information Hiding
  - -- Abstract Data Types

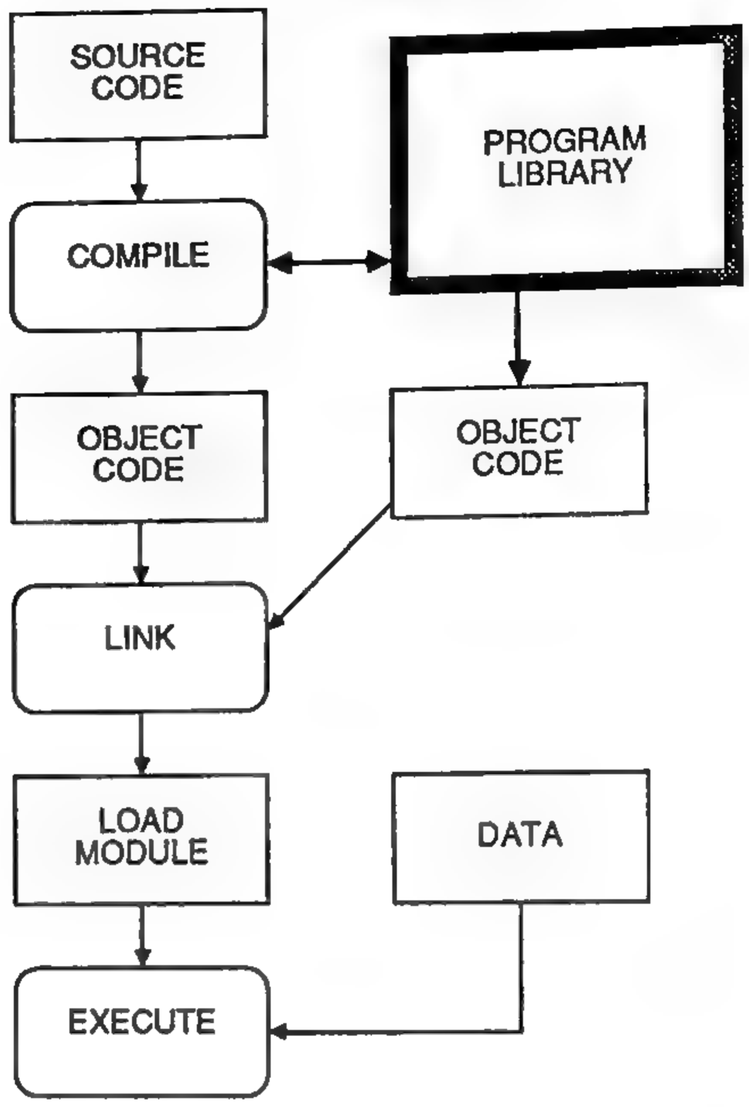


- GENERICS
  - -- Packages and subprograms
  - -- HOL macro

TRADITIONAL



Ada approach

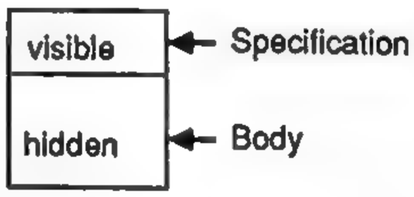


ALL Ada PROGRAM UNITS

- The SPECIFICATION (outside view) is the contract or interface between the user of the unit and the implementor of the unit. It represents only "What" is to be done, not "how".

- The BODY (inside view) is the "how" of the unit. Its details are the responsibility of the implementor. The user of the unit need not (and should not) know these details.

Ada SUBPROGRAMS



- PROCEDURES
  - -- Perform some "sub-action"
  - -- Call always appears as a statement
- FUNCTIONS
  - -- Calculate and return a value
  - -- Call always appears in an expression

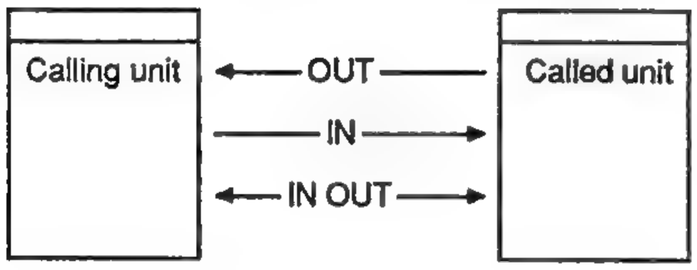
PARAMETER PASSING MODES

IN - The formal parameter acts as a local constant. Assignment (definition) is not allowed.

OUT - The formal parameter holds a 'created' value. Reference is not allowed.

IN OUT - The formal parameter can be both assigned to (defined) and referenced.

- The default mode is IN
- Functions may have IN parameters only



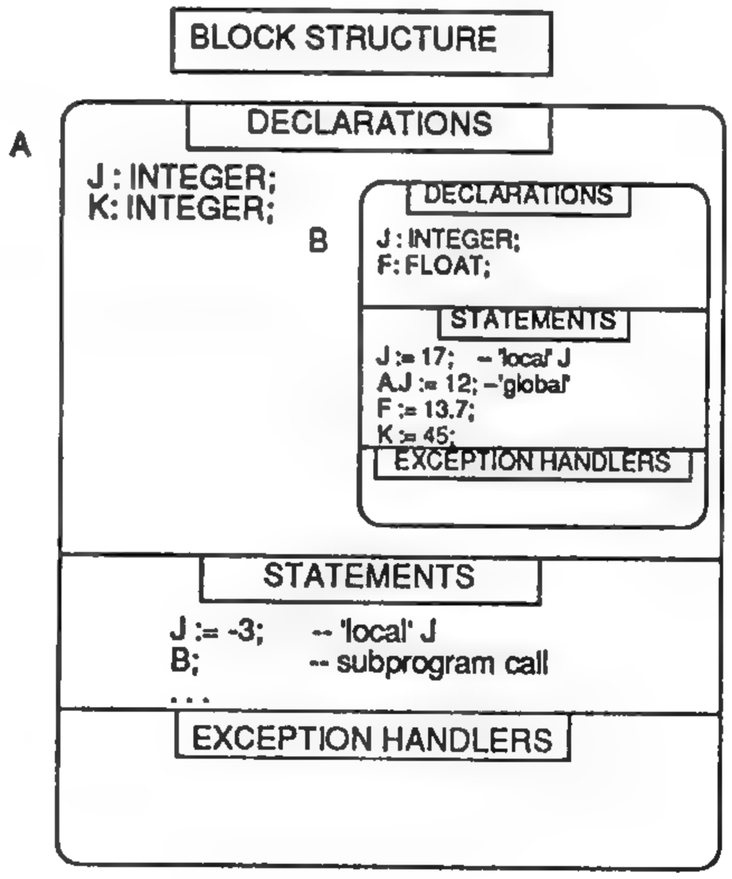
Ada PROCEDURES

```
-- PROCEDURE SPECIFICATION
procedure SWAP (PRE, POST : in out INTEGER);

-- PROCEDURE CALL
SWAP (MY_COUNT, YOUR_COUNT);
SWAP (PRE => MY_COUNT, POST => YOUR_COUNT);
SWAP (POST => YOUR_COUNT, PRE => MY_COUNT);

-- PROCEDURE BODY
procedure SWAP (PRE, POST : in out INTEGER) is
   TEMP : INTEGER := PRE;     -- local object declaration
begin
   PRE  := POST;
   POST := TEMP;
end SWAP;
```

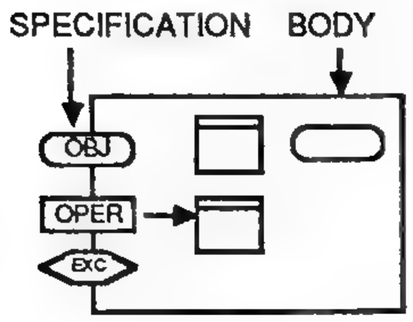

Ada FUNCTIONS

```
-- FUNCTION SPECIFICATION

function SQRT (ARG : FLOAT) return FLOAT;

-- FUNCTION CALL

-- assuming STANDARD_DEV and VARIANCE are
-- of type float:

STANDARD_DEV := SQRT (VARIANCE);

-- FUNCTION BODY

function SQRT (ARG :FLOAT) return FLOAT is
     RESULT : FLOAT;
begin
     -- algorithm for computing RESULT goes here
     return RESULT;
end SQRT;
```


BLOCK STRUCTURE



Ada PACKAGES



- The PACKAGE is the primary means of "extending" the Ada language
- The PACKAGE hides information in the body thereby enforcing the abstraction represented by the specification
- Operations (subprograms, functions etc.) whose specification appear in the package specification must have their body appear in the package body.
- Other units (subprograms, functions, packages etc.) as well as other types, objects etc. may also appear in the package body. If so, they are not visible outside the package body.

Ada PACKAGES

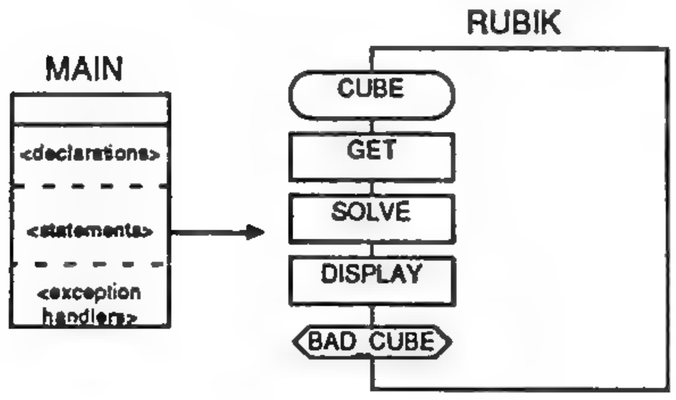
OUTSIDE

```
-- PACKAGE SPECIFICATION

package RUBIK is

   type CUBE is private;
   procedure GET (C : out CUBE);
   procedure SOLVE (C : in out CUBE);
   procedure DISPLAY (C : in CUBE);
   BAD_CUBE : exception;

private

   type CUBE is . . .  -- Actual type definition goes here

end RUBIK;
```

INSIDE

```
-- PACKAGE BODY

package body RUBIK is

        -- all bodies of subprograms found in the
        -- package spec go here along with any
        -- other local declarations that should
        -- be kept "hidden" from the user.

   procedure GET (C : out CUBE) is . . .

   procedure SOLVE (C : in out CUBE) is . . .

   procedure DISPLAY (C : in CUBE) is . . .

end RUBIK;
```


PACKAGE USAGE

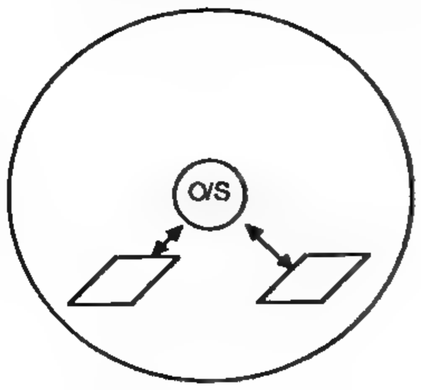



```
with RUBIK, TEXT_IO;
procedure MAIN is

    MY_CUBE : RUBIK.CUBE;

begin

    RUBIK.GET (MY_CUBE);
    RUBIK.SOLVE (MY_CUBE);
    RUBIK.DISPLAY (MY_CUBE);

exception

    when RUBIK.BAD_CUBE =>
       TEXT_IO.PUT_LINE ("You got a bad one");

end MAIN;
```

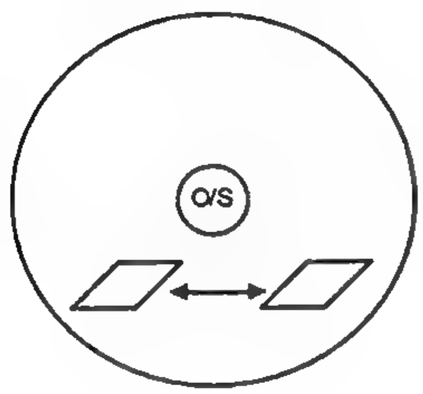


```
Package MEASURES is          -- specification
    type AREA   is private;
    type LENGTH is private;
    ...
    function "+" (LEFT, RIGHT : LENGTH) return LENGTH;
    function "*" (LEFT, RIGHT : LENGTH) return AREA;
    ...
    NUMBER_TOO_LARGE : exception;
private
    type AREA is range 0 . . 10000;
    type LENGTH is range 0 . . 100;
end MEASURES;

-------------------------------------------------

with MEASURES;
procedure MEASUREMENT is

    SIDE_1, SIDE_2 : MEASURES.LENGTH;
    FIELD : MEASURES.AREA;

    use MEASURES;        -- allow direct visibility

begin
    ...
    FIELD := SIDE_1 * SIDE_2;
    ...
exception

    when NUMBER_TOO_LARGE => . . .

end MEASUREMENT;
```

THE ULTIMATE IN INFORMATION HIDING

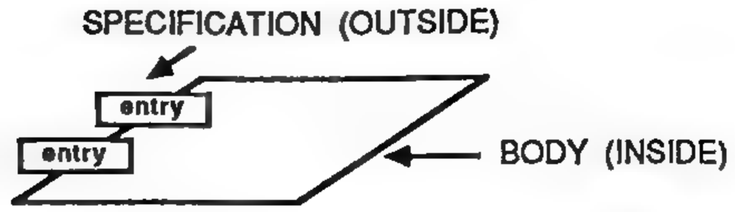


THE TRADITIONAL MODEL OF CONCURRENCY



THE Ada TASKING MODEL

Ada TASKS



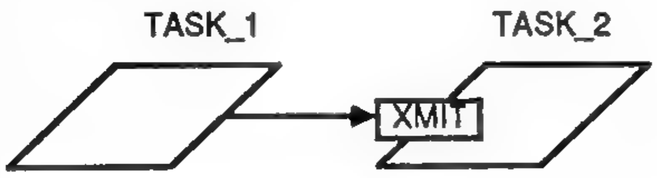
- The TASK concept in Ada provides a model of parallelism which encompasses:

  -- Multicomputers
  -- Multiprocessors
  -- Interleaved Execution

- In Ada, the method of communication between tasks is known as "rendezvous"

- Ada "draws up" into the language certain capabilities previously performed only by the operating system

TASK COMMUNICATION

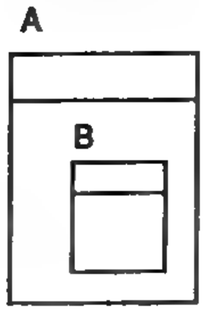



```
-- TASK SPECIFICATIONS

        task TASK_1;     -- no entries

        task TASK_2 is
           entry XMIT (N : in INTEGER);
        end  TASK_2;

-- TASK BODIES

        task body TASK_1 is
           ...
           TASK_2.XMIT (17);   -- an entry call
           ...
        end TASK_1;

        task body TASK_2 is
           ...
           accept XMIT (N : in INTEGER) do
           ...          -- statements to be executed
                        -- during rendezvous
           end XMIT;
           ...
        end TASK_2;
```

SEPARATE COMPILATION

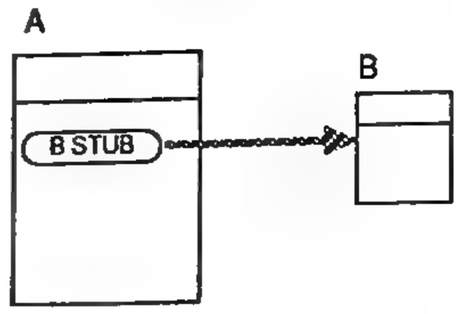
- PACKAGE, TASK AND SUBPROGRAM BODIES CAN BE COMPILED SEPARATELY FROM THEIR SPECIFICATIONS
- THE INDICATOR OF SEPARATE COMPILATION IS KNOWN AS A 'STUB'
- THE SEPARATELY COMPILED BODY IS KNOWN AS A 'SUBUNIT'
- THE UNIT WHICH CONTAINS THE 'STUB' IS KNOWN AS THE 'PARENT'
- ENTITIES VISIBLE TO THE 'STUB' ARE ALSO VISIBLE TO THE 'SUBUNIT'

TEXTUALLY NESTED




```
procedure A ( ) is
   ...
   procedure B ( ) is
      ...
   begin
      ...
   end B;
begin
   ...
end A;
```

SEPARATELY COMPILED




```
procedure A ( ) is
   ...
   procedure B ( ) is separate;
begin
   ...
end A;
```

```
separate (A)
procedure B( ) is
   ...
begin
   ...
end B;
```


COMPILATION UNIT DEPENDENCIES

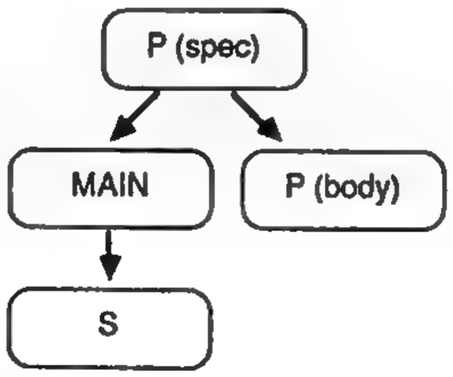
1. PARENT UNITS ARE COMPILED BEFORE THEIR SUBUNITS (Recompiling the parent requires recompiling the subunit)

2. SPECIFICATIONS ARE COMPILED BEFORE THEIR BODIES (Recompiling the specification requires recompiling the body)

3. REFERENCED LIBRARY UNITS ARE COMPILED BEFORE ANY UNITS WHICH REFERENCE THEM (Recompiling the referenced unit requires recompiling the unit which references it)

```
with P;
procedure MAIN is
   ...
   procedure S is separate;
begin
   ...
end MAIN;
```

```
separate (MAIN)
procedure S is
   ...
begin
   ...
end S;
```



Ada FROM THE BOTTOM UP

- CHARACTER SET
  - -- All Ada constructs are built from the ASCII character set
- LEXICAL UNITS

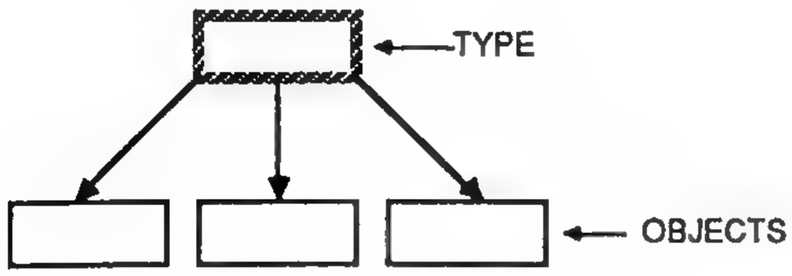
-- Identifiers	(COUNT, begin)
-- Numeric Literals	(17, 3.5, 8#77#)
-- Character Literals	('A', 'a', ' ', '5', '"')
-- Strings	("This is a string")
-- Delimiters	(&, +, :, <>, =>)
-- Comments	

Ada RESERVED WORDS

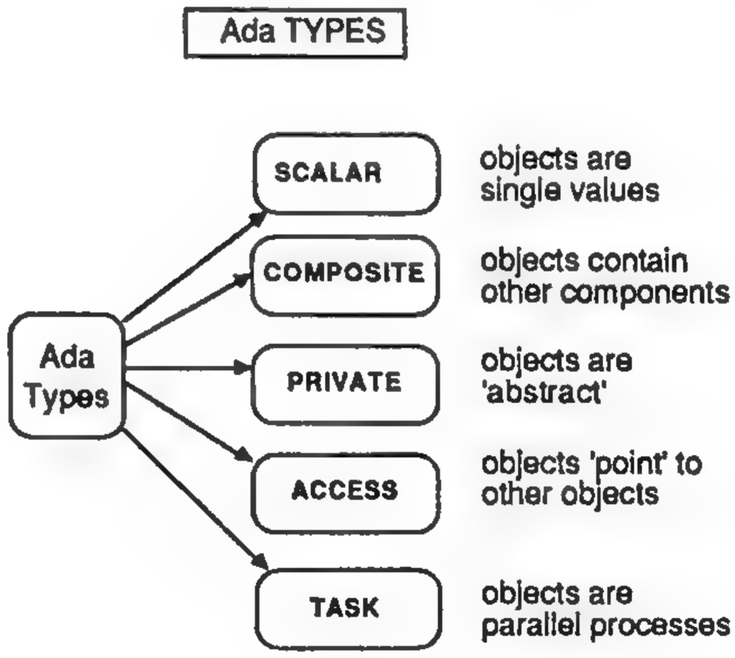
abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits		out	
all	do	if		
and		in		
array		is	package	task
at	else		pragma	terminate
	elsif	limited	private	then
	end	loop	procedure	type
	entry			
	exception			
begin	exit	mod	raise	use
body			range	
			record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

Ada TYPES

- A Type is a template for objects; it represents a set of values which are meaningful for the objects and also a set of operations on the objects (values)
- Ada is a strongly typed language. This means that all objects must be declared and objects of different types cannot be implicitly mixed in operations
- TYPES are not operated upon directly. They are a means of declaring instances called OBJECTS. These objects can be operated upon.



Ada TYPES



SCALAR TYPES			
DISCRETE		REAL	
INTEGER	ENUMERATED	FIXED	FLOAT
integer natural positive	boolean character	duration	float

← USER DEFINED →

ENUMERATION TYPE DECLARATIONS

```
type COLOR is (WHITE, RED, YELLOW, GREEN, BLUE);
type LIGHT  is (RED, AMBER, GREEN);
type GEAR_POSITION is (UP, DOWN, NEUTRAL);
type SUITS is (CLUBS, DIAMONDS, HEARTS, SPADES);
subtype MAJORS is SUITS range HEARTS . . SPADES;
type BOOLEAN is (FALSE, TRUE);  -- predefined
```

ENUMERATION OBJECT DECLARATIONS

```
HUE : COLOR;
SHIFT : GEAR_POSITION := GEAR_POSITION'LAST;
T : constant BOOLEAN := TRUE;
HIGH : MAJORS := CLUBS;  -- invalid
```

HUE	SHIFT	T
undefined	NEUTRAL	TRUE

COMPOSITE TYPES

ARRAY TYPES DESCRIBE COLLECTIONS OF HOMOGENEOUS COMPONENTS. INDIVIDUAL COMPONENTS ARE SELECTED BY DISCRETE INDEX.

RECORD TYPES DESCRIBE COLLECTIONS OF HETEROGENEOUS COMPONENTS. INDIVIDUAL COMPONENTS ARE SELECTED BY FIELD IDENTIFIER.

CONSTRAINED ARRAYS

```
type TABLE is array (INTEGER range 1 .. 5) of FLOAT;
MY_LIST : TABLE := (3.7,  14.2,  -6.5,  0.0,  1.0);

type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);
type WEEK_ARRAY is array (DAYS) of BOOLEAN;

T : constant BOOLEAN := TRUE;
F : constant BOOLEAN := FALSE;

MY_WEEK : WEEK_ARRAY := (MON .. FRI => T, others => F);
```

MY_LIST

1	3.7
2	14.2
3	-6.5
4	0.0
5	1.0

MY_WEEK

SUN	F
MON	T
TUE	T
WED	T
THU	T
FRI	T
SAT	F

```
MY_LIST (4) := 7.3;
if MY_WEEK (THU) = true then ...
if MY_WEEK (THU)  then ...
```

UNCONSTRAINED ARRAYS

- INDEX TYPE AND COMPONENT TYPE BOUND TO ARRAY TYPE
- INDEX RANGE BOUND TO OBJECTS, NOT TYPE
- ALLOWS FOR GENERAL PURPOSE SUBPROGRAMS

```
type SAMP is array (INTEGER range <>) of FLOAT;

LARGE : SAMP (1 .. 5) := (2.5, 3.4, 1.0, 0.0, 4.4);
SMALL : SAMP (2 .. 4) := (2 .. 4 => 5.0);
```

LARGE

1	2.5
2	3.4
3	1.0
4	0.0
5	4.4

SMALL

2	5.0
3	5.0
4	5.0

RECORD TYPES

```
-- Record type declaration

type DATE is
  record
    DAY   : INTEGER range 1 .. 31;
    MONTH : MONTH_TYPE;
    YEAR  : INTEGER range 1700 .. 2150
  end record;

-- Record object declaration

TODAY : DATE;

-- Record component reference

TODAY.DAY   := 4;
TODAY.MONTH := JUL;
TODAY.YEAR  := 1776;

-- Record object reference

TODAY := (4, JUL, 1776);

-- or --

if TODAY /= (6, DEC, 1942) then . . .
```

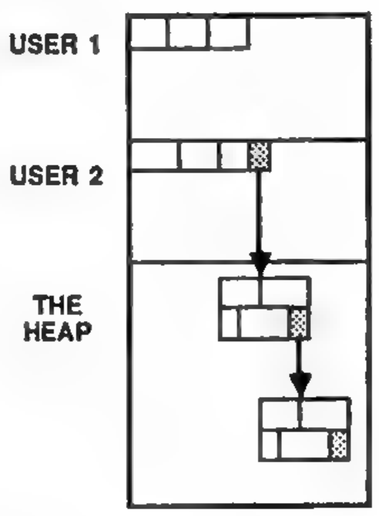
TODAY

DAY
MONTH
YEAR

ACCESS TYPES

```
type NODE;

type PTR is access NODE;

type NODE is
   record
      FIELD_1: SOME_TYPE;
      FIELD_2: BLAH;
      FIELD_3: FOO;
      FIELD_4: FRAMUS;
      FIELD_5: PTR;
   end record;
```

(Memory Allocation)




```
TOP : PTR;  -- an access object
. . .
TOP := new NODE;  -- an allocator

TOP.FIELD_5 := new  NODE;  -- another allocator
```


Ada Statements

SEQUENTIAL

- ASSIGNMENT
- NULL
- PROCEDURE CALL
- RETURN
- BLOCK

CONDITIONAL

- IF
- --THEN
- --ELSE
- --ELSIF
- CASE

ITERATIVE

- LOOP
- --EXIT
- --FOR
- --WHILE

TASKING

- DELAY
- ENTRY CALL
- ABORT
- ACCEPT
- SELECT

OTHER

- RAISE
- CODE
- goto

Ada STATEMENTS

```
-- To exemplify some of the Ada statements,
-- consider the implementation of a 'wrap-around'
-- successor function for type DAYS.

procedure TEST is
    type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);
    TODAY, TOMORROW : DAYS;
     function WRAP (D :  DAYS) return DAYS is . . .
begin
  . . .
  TOMORROW := WRAP (TODAY);
  . . .
end TEST;
```



```
function WRAP (D : DAYS) return DAYS is
begin
        if D = SUN then
            return MON;
        elsif D = MON then
            return TUE;
        elsif D = TUE then
            return WED;
        elsif D = WED then
            return THU;
        elsif D = THU then
            return FRI;
        elsif D = FRI then
            return SAT;
        else
            return SUN;
        end if;
end WRAP;
```



```
function WRAP (D : DAYS) return DAYS is
begin
        case D is
            when SUN => return MON;
            when MON => return TUE;
            when TUE  => return WED;
            when WED => return THU;
            when THU  => return FRI;
            when FRI   => return SAT;
            when SAT   => return SUN;
        end case;
end WRAP;
```



```
function WRAP (D : DAYS) return DAYS is
        WEEK : array (DAYS) of DAYS :=
            (MON, TUE, WED, THU, FRI, SAT, SUN);
begin
        return WEEK (D);
end WRAP;
```

	WEEK
SUN	MON
MON	TUE
TUE	WED
WED	THU
THU	FRI
FRI	SAT
SAT	SUN

```
function WRAP (D : DAYS) return DAYS is
begin
        return DAYS'SUCC (D);
exception
        when CONSTRAINT_ERROR =>
          return DAYS'FIRST;
end WRAP;
```



```
function WRAP (D : DAYS) return DAYS is
begin
        if D = SAT then
            return SUN;
        else
            return DAYS'SUCC(D);
        end if;
end WRAP;
```



```
function WRAP (D : DAYS) return DAYS is
begin
        if D = DAYS'LAST then
            return DAYS'FIRST;
        else
            return DAYS'SUCC (D);
        end if;
end WRAP;
```


Consider the following integer type declaration:

type SIZE is range 1 . . 10;

Suppose you wanted a wrap-around successor capability for this type. That is, the successor of the value 10 would be the value 1.

What changes would need to be made to the previous example in order to provide this capability?

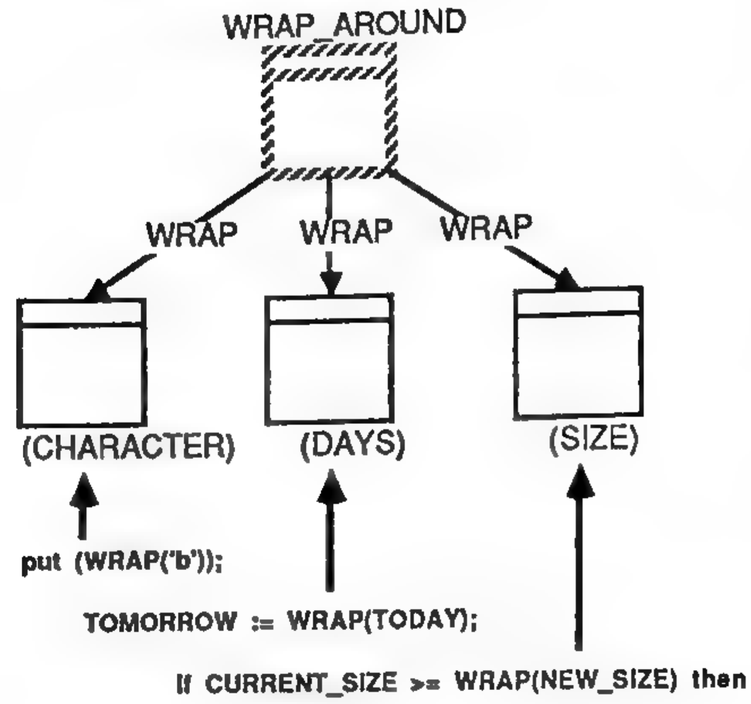
GENERIC UNITS

GENERIC SPECIFICATION

```
generic
      type ELEMENT is (<>);
function WRAP_AROUND (D : ELEMENT) return ELEMENT;
```

GENERIC BODY

```
function WRAP_AROUND (D : ELEMENT) return ELEMENT is
begin
      if D = ELEMENT'LAST then
         return ELEMENT'FIRST;
      else
         return ELEMENT'SUCC (D);
      end if;
end WRAP_AROUND;
```

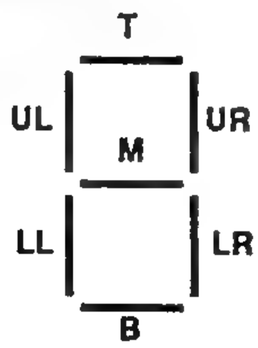
GENERIC INSTANTIATION

```
function WRAP is new WRAP_AROUND (ELEMENT => DAYS);
function WRAP is new WRAP_AROUND (ELEMENT => SIZE);
function WRAP is new WRAP_AROUND (CHARACTER);
```

-- NOTE: The identifiers of the instantiations need not be overloaded

A package for dealing with digital representations of numbers:

```
package DIGITAL_INFO is

   type LIGHT_POSITION is (T, UL, UR, M, LL, LR, B);
   type LIGHT_STATUS   is (OFF, ON);

   type DIGITAL_VALUE is array (LIGHT_POSITION)
                                 of LIGHT_STATUS;

   type DECIMAL  is range 0 .. 9;

   function CONVERT (NUM : DECIMAL)
                     return DIGITAL_VALUE;

   -- other resources could go here
end DIGITAL_INFO;
```

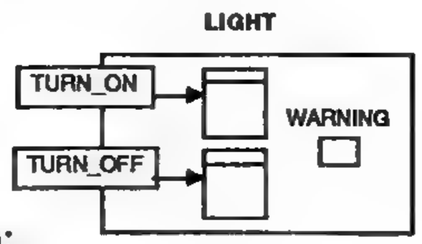


T	ON
UL	OFF
UR	ON
M	ON
LL	OFF
LR	ON
B	ON


```
package body DIGITAL_INFO is

   function CONVERT (NUM : DECIMAL) return DIGITAL_VALUE is
   begin

      case NUM is

         when 0 => return (M => OFF,          others => ON);
         when 1 => return (UR | LR => ON,     others => OFF);
         when 2 => return (UL | LR => OFF,    others => ON);
         when 3 => return (UL | LL => OFF,    others => ON);
         when 4 => return (T | LL | B => OFF, others => ON);
         when 5 => return (UR | LL => OFF,    others => ON);
         when 6 => return (UR => OFF,         others => ON);
         when 7 => return (T | UR | LR => ON, others => OFF);
         when 8 => return (                   others => ON);
         when 9 => return (LL | B => OFF,     others => ON);

      end case;

   end CONVERT;

      -- bodies of other units go here

end DIGITAL_INFO;
```

REPRESENTATION SPECIFICATIONS

Allow the user to turn a warning light on and off. The light is mapped into HEX location 100. If the first eight bits of that location are set to all ones, the light will be on. If the first eight bits are set to all zeroes, the light will be off. There are no guarantees relative to any other configuration.




```
package LIGHT is
    procedure TURN_ON;
    procedure TURN_OFF;
end LIGHT;

package body LIGHT is

    type STATUS is (OFF, ON);
    for STATUS'SIZE use 8;
    for STATUS use (OFF => 16#00#,
                    ON  => 16#FF#);
    WARNING : STATUS := OFF;
    for WARNING use at 16#100#;

    procedure TURN_ON is
    begin
        WARNING := ON;
    end TURN_ON;

    procedure TURN_OFF is
    begin
        WARNING := OFF:
    end TURN_OFF;

end LIGHT;
```


DESIGN EXAMPLE

COUNT THE NUMBER OF WORDS IN EACH OF A SEQUENCE OF TELEGRAMS.

(From George Cherry's book "Parallel Programming In ANSI Standard Ada")

An input file contains the text of a number of telegrams. Each telegram consists of a number of words followed by the word "ZZZZ".

The input file is composed of a sequence of lines. The lines can vary in length; but the length of a line cannot exceed 40 characters. Each line contains a number of words, separated by blanks.

The length of a word cannot exceed 26 characters. There may be one or more blanks between adjacent words; and there may be one or more additional blanks at the beginning and end of a line.

There is no particular relationship between telegrams and lines: a telegram may begin and end anywhere in a line and may span several lines. Furthermore, several telegrams may share a line.

The problem is to analyze the set of telegrams and print a report, showing for each telegram its ordinal number and the number of words it contains. Of course, the special "word" "ZZZZ" should not be counted as a word in the statistics.

INPUT

DEAR MOM: HAPPY BIRTHDAY! I LOVE, TIM ZZZZ
DAD: SEND MONEY. JOE ZZZZ MR. PRESIDENT:
PLEASE RESTORE THE BUDGET FOR STARS.
VANCE DRUFFEL ZZZZ DEAR ELIZABETH: BEST
WISHES ON YOUR LATEST MATRIMONIAL TRY.
J. WARNER ZZZZ DEAR J. GO TO H---I E. T.
ZZZZ DEAR GEORGE: GO FOR IT! J. I. ZZZZ
DEAR JEAN: ROSES ARE RED; VIOLETS ARE
BLUE; ADA IS GREEN. D. F. ZZZZ DEAR 007:
009 HAS BEEN ASSASSINATED; YOUR NEW
CONTACT IS 008. CONTROL ZZZZ

OUTPUT

```
Telegram number  1 contains  6 words.
Telegram number  2 contains  4 words.
Telegram number  3 contains 10 words.
Telegram number  4 contains 11 words.
Telegram number  5 contains  7 words.
Telegram number  6 contains  7 words.
Telegram number  7 contains 13 words.
Telegram number  8 contains 12 words.
```

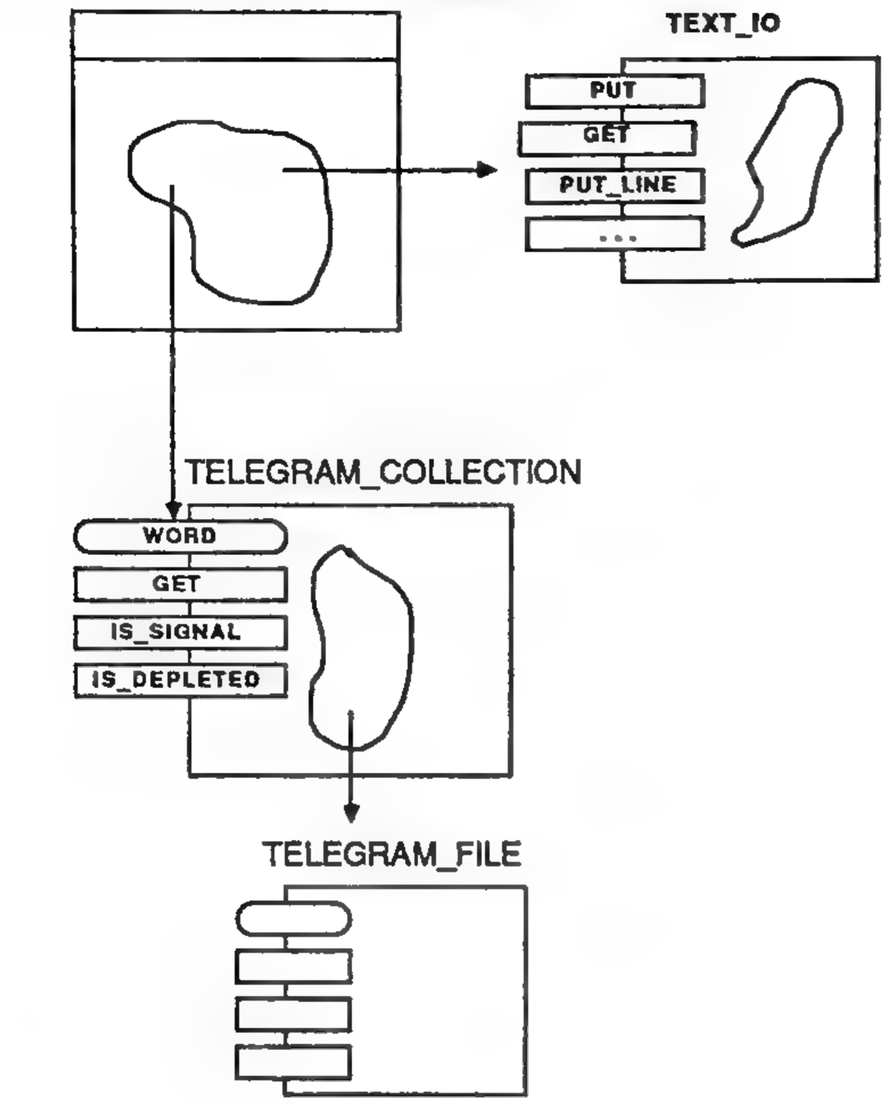
END OF REPORT

INFORMAL STRATEGY

A COLLECTION OF TELEGRAMS IS A SEQUENCE OF WORDS WITH SPECIAL SIGNAL WORDS INSERTED AT THE END OF EACH TELEGRAM. WHILE WORDS REMAIN, GET A WORD AND, IF IT IS NOT A SIGNAL WORD, INCREMENT THE COUNTER ASSOCIATED WITH THE TELEGRAM. IF THE WORD IS A SIGNAL WORD, OUTPUT THE COUNT OF WORDS AND CLEAR THE COUNTER. WHEN THERE ARE NO MORE WORDS, OUTPUT AN APPROPRIATE MESSAGE.

OBJECTS AND OPERATIONS

TELEGRAM COLLECTION
-- GET NEXT WORD
-- WORD IS SIGNAL
-- COLLECTION IS DEPLETED



OBJECT SPECIFICATION

```
package TELEGRAM_COLLECTION is

    type WORD is private;

    procedure GET (THE_WORD : out WORD);

    function IS_DEPLETED return BOOLEAN;

    function IS_SIGNAL (THE_WORD : WORD)
                                    return BOOLEAN;

private

    type WORD is . . .

end TELEGRAM_COLLECTION;
```



```
with TELEGRAM_COLLECTION, TEXT_IO;
procedure  COUNT_WORDS_IN_TELEGRAMS  is

   TELEGRAM_NUMBER  : NATURAL := 0;
   WORD_COUNT       : NATURAL := 0;
   CURRENT_WORD     : TELEGRAM_COLLECTION.WORD;

      package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

      procedure OUTPUT_COUNT (NUMBER, COUNT : NATURAL)
                              is separate;

begin

   loop

         exit when TELEGRAM_COLLECTION.IS_DEPLETED;

        TELEGRAM_COLLECTION.GET(CURRENT_WORD);

         if TELEGRAM_COLLECTION.IS_SIGNAL(CURRENT_WORD) then

           TELEGRAM_NUMBER := TELEGRAM_NUMBER + 1;
           OUTPUT_COUNT  (TELEGRAM_NUMBER,  WORD_COUNT);
           WORD_COUNT := 0;

       else

            WORD_COUNT := WORD_COUNT + 1;

       end if;

    end loop;

    TEXT_IO.PUT_LINE("              END OF REPORT");

end  COUNT_WORDS_IN_TELEGRAMS;
```



```
separate  (COUNT_WORDS_IN_TELEGRAMS)
procedure OUTPUT_COUNT (NUMBER, COUNT : in NATURAL) is

begin
    TEXT_IO.PUT ("Telegram number");

    INT_IO.PUT   (NUMBER,2);

    TEXT_IO.PUT (" containe ");

    INT_IO.PUT   (COUNT,2);

    TEXT_IO.PUT (" words.");

end  OUTPUT_COUNT;
```

DATA TYPES

- A TYPE CHARACTERIZES A SET OF VALUES WHICH OBJECTS OF THE TYPE CAN TAKE ON AND A SET OF VALID OPERATIONS ON THE OBJECTS
- TWO DIFFERENT TYPE DECLARATIONS ALWAYS DEFINE TWO DISTINCT TYPES
- OBJECTS OF DISTINCT TYPES CANNOT BE OPERATED UPON TOGETHER WITHOUT EXPLICIT CONVERSION

TYPE DECLARATIONS

type RR_CARS is 

type ANIMALS is 

type MIXED is 

OBJECT DECLARATIONS

CAR : RR_CARS := 

SPOT : MIXED := 

PHYDEAUX : constant ANIMALS :=

CAR	SPOT	PHYDEAUX
		

Which of the following are valid assignment statements?

1. CAR :=
2. SPOT :=
3. PHYDEAUX :=
4. SPOT := PHYDEAUX;
5. PHYDEAUX := SPOT;
6. CAR :=
7. SPOT :=
8. SPOT :=

SUBTYPES

- A SUBTYPE IS A TYPE TOGETHER WITH A CONSTRAINT
- THE TYPE IS KNOWN AS A BASE TYPE
- THE CONSTRAINT CAN BE NULL (an alias)
- A TYPE IS A SUBTYPE OF ITSELF
- A VALUE BELONGS TO A SUBTYPE OF A GIVEN TYPE IF IT BELONGS TO THE TYPE AND SATISFIES THE CONSTRAINT
- THE SUBTYPE INHERITS ALL OPERATIONS FROM THE BASE TYPE
- A TYPE MARK IS A TYPE IDENTIFIER OR A SUBTYPE IDENTIFIER
- THE TYPE OF AN OBJECT IS KNOWN AT COMPILATION TIME static
- VIOLATION OF SUBTYPE IS ALWAYS A CONSTRAINT ERROR dynamic

objects of a subtype are implicitly compatible with objects of the base type and with objects of other subtypes with the same base type

type THINGS is

subtype WEAPONS is THINGS THRU

subtype POINTED_OBJECTS is THINGS THRU

OBJECT DECLARATIONS

```
LETHAL : WEAPONS := 
MY_OBJECT : THINGS := 
SHARP : POINTED_OBJECTS := 
```


Which of the following are valid assignment statements?

1. MY_OBJECT :=
2. MY_OBJECT := SHARP;
3. MY_OBJECT := LETHAL;
4. LETHAL :=
5. LETHAL := MY_OBJECT;
6. LETHAL :=
7. SHARP := LETHAL;
8. SHARP := MY_OBJECT;

INTEGER TYPES

- An Integer type characterizes a set of whole number values and a set of operations on whole numbers

```
type DEPTH is range -1000 .. 0;
type ROWS is range 1 .. 8;
type LINES is range 0 .. 66;

subtype TERMINAL is LINES range 0 .. 24;
```

INTEGER OBJECT DECLARATIONS

```
ROW_COUNT  : ROWS;
LINE_COUNT : LINES := 1;
CRT        : TERMINAL := 16;
FATHOMS    : constant DEPTH := -100;
```

ROW_COUNT	LINE_COUNT	CRT	FATHOMS
undefined	1	16	-100

INTEGER ATTRIBUTES

```
type SAMPLE is range 1 .. 20;
```

- SAMPLE'FIRST -- 1
- SAMPLE'LAST -- 20
- SAMPLE'PRED (17) -- 16
- SAMPLE'SUCC (20) -- CONSTRAINT_ERROR
- SAMPLE'IMAGE (12) -- "12"
- SAMPLE'VALUE ("12") -- 12
- SAMPLE'VALUE ("21") -- CONSTRAINT_ERROR

```
MY_INT : SAMPLE := SAMPLE'FIRST;
```

- 'Based Literals' explicitly specify the base from two to sixteen
- 'Extended Digits' are the letters 'A' thru 'F'

```
MY_HEX_VALUE : NATURAL := 16#7A8#;
MY_OCTAL : NATURAL := 8#7773#E2;
THIRTY_ONE : constant INTEGER := 2#1_1111#;
```

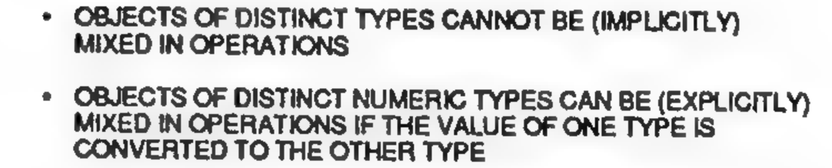
REAL TYPES

- Real types provide approximations to the real numbers.
- There is always some error associated with a value of a real type.
- If the error grows as the magnitude of the number increases then we are dealing with floating point types (relative precision).
- If the error remains constant as the magnitude of the number increases then we are dealing with fixed point types (absolute precision).
- A real type determines a set of model numbers which can be represented exactly.
- If an operation yields a model number, it delivers that number. If it yields a number between two model numbers, it delivers either the lower or upper.

FIXED POINT TYPES

- INDICATES ACTUAL DIFFERENCE BETWEEN MODEL NUMBERS
- RANGE CONSTRAINT IS NOT OPTIONAL FOR TYPE
- RANGE CONSTRAINT IS OPTIONAL FOR SUBTYPE

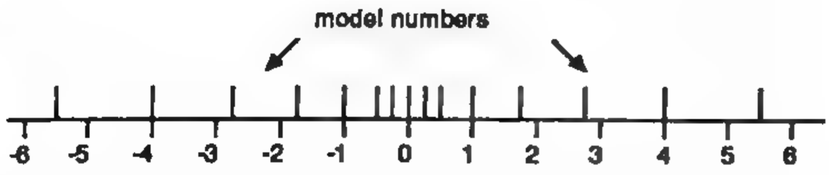
```
type MONEY is delta 0.01 range 0.0 .. 1_000_000.0;
subtype PAY is   MONEY range 0.0 .. 1_000.0;
subtype DOLLARS is MONEY delta 1.0;
```



FLOATING POINT TYPES

- INDICATES NUMBER OF SIGNIFICANT DIGITS (actually converted to significant bits)
- THE TYPE IS GUARANTEED TO HAVE AT LEAST THIS MUCH PRECISION
- AN IMPLEMENTATION WILL REPORT IF IT IS UNABLE TO HANDLE THE REQUESTED PRECISION
- RANGE CONSTRAINT IS OPTIONAL

```
type COEFFICIENT is digits 10 range -1.0 .. 1.0;
type REAL is digits 8;
subtype SHORT_COEFF is COEFFICIENT digits 5;
subtype NARROW is REAL range 0.0 .. 20.0;
```



TYPE CONVERSION FUNCTIONS

- OBJECTS OF DISTINCT TYPES CANNOT BE (IMPLICITLY) MIXED IN OPERATIONS
- OBJECTS OF DISTINCT NUMERIC TYPES CAN BE (EXPLICITLY) MIXED IN OPERATIONS IF THE VALUE OF ONE TYPE IS CONVERTED TO THE OTHER TYPE
- THE IDENTIFIER OF THE TYPE BECOMES THE IDENTIFIER OF A FUNCTION FOR PURPOSES OF CONVERSION (TRANSFER)

```
type MY_INT  is range 0 .. 100;
type MY_FLT is digits 10 range 0.0 .. 100.0;

INT_OBJECT : MY_INT;
FLT_OBJECT : MY_FLT;

...

INT_OBJECT   :=  MY_INT (FLT_OBJECT);   -- rounding

FLT_OBJECT   :=  MY_FLT (INT_OBJECT);
```

type transfer

EXPONENTIATION

X ** Y

if X is of any Integer type then Y must be of the predefined type INTEGER and must not be negative.

if X is of any real type then Y must be of the predefined type INTEGER.

The above two rules apply only for the exponentiation operation which is implicit with a type. The programmer is free to overload the operator to provide exponentiation by values other than INTEGER.

NUMBER DECLARATIONS

- A SPECIAL FORM OF CONSTANT DECLARATION
- THE EXPRESSION MUST BE STATIC AND EITHER

  universal_integer or

  universal_real

- INTEGER NAMED NUMBERS ARE IMPLICITLY COMPATIBLE WITH ANY INTEGER TYPE
- REAL NAMED NUMBERS ARE IMPLICITLY COMPATIBLE WITH ANY REAL (FIXED OR FLOAT) TYPE

```
PI              : constant := 3.14159_26536;
TWO_PI          : constant := 2.0 * PI;
MAX             : constant := 500;
POWER_16        : constant := 2 ** 16;
ONE, UN, EINS   : constant := 1;
```


ENUMERATION TYPE ATTRIBUTES

```
type SPEED is (SLOW, MODERATE, FAST);
                0      1       2
SPEED'FIRST             -- SLOW
SPEED'LAST              -- FAST
SPEED'SUCC(SLOW)        -- MODERATE
SPEED'PRED(SLOW)        -- CONSTRAINT_ERROR
SPEED'POS(SLOW)         -- 0
SPEED'VAL(2)            -- FAST
SPEED'IMAGE(FAST)       -- "FAST"
SPEED'VALUE("SLOW")     -- SLOW
SPEED'VALUE("slow")     -- SLOW
SPEED'VALUE("QUICK")    -- CONSTRAINT_ERROR
SPEED'WIDTH             -- 8
```

max length of image

subtype is contiguous range

CHARACTER TYPE DECLARATIONS

```
type CHARACTER is (nul, soh, .....'A'.....'a'.....) -- predefined
type ROMAN_DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
type VOWELS is ('A', 'E', 'I', 'O', 'U');
subtype FORTRAN_CONVENTION is CHARACTER range 'I' . . 'N';
```

CHARACTER OBJECT DECLARATIONS

```
INDEX : FORTRAN_CONVENTION := 'K';
ROMAN_100 : constant ROMAN_DIGIT := 'C';
MY_ CHAR : CHARACTER;
```

INDEX	ROMAN_100	MY_CHAR
'K'	'C'	*undefined*

NOTE: In Ada, character types are considered to be enumerated types. This is not the case in Pascal.

TYPE BOOLEAN

```
type BOOLEAN is (FALSE, TRUE);   --   predefined

P, Q, R : BOOLEAN;


-- All relational operators apply (=, /=, <, <=, >, >=)
-- The following logical operators are in the language:
     NOT, AND, OR, XOR


P or Q or R          -- a legal boolean expression
P and Q and R        -- also legal
P or Q and R         -- illegal, needs parentheses
P or (Q and R)       -- legal
(P or Q) and R       -- legal
```


HIERARCHY OF OPERATIONS

- Highest Precedence ** NOT ABS
- Multiplicative * / MOD REM
- Unary Additive + -
- Binary Additive + - &
- Relational = /= < > <= >=
- Membership IN NOT IN
- Logical AND OR XOR
- Short-Circuit AND THEN OR ELSE

MEMBERSHIP OPERATION

- Used to determine if an expression is in a given subtype
- Expression must be of the same basetype as the subtype
- Result of IN is true if the expression is in the subtype
- NOT IN is an infix operation
- Membership operations cannot be overloaded

```
subtype ALPHA is CHARACTER range 'A' .. 'Z';
CH : CHARACTER;
NUM : INTEGER;
. . .
TEXT_IO.GET(CH);
if CH in ALPHA then . . .

if NUM in 7 .. 15 then . . .
```

The following are equivalent:

```
. . . CH not in ALPHA. . .
. . .not (CH in ALPHA). . .
```


SHORT-CIRCUIT OPERATORS

A	B	not A	A and B	A or B	A xor B
T	T	F	T	T	F
T	F	F	F	T	T
F	T	T	F	T	T
F	F	T	F	F	F

Both subexpression for AND, OR and XOR will always be evaluated.

AND THEN and OR ELSE are operations which will evaluate the right hand side of a boolean expression only if the left hand side has not already determined the result of the expression

```
if X /= 0   and then   Y/X >= 17 then . . .

if PTR = null   or else   PTR.LEFT > 10 then . . .
```

CONSTRAINED ARRAYS

```
type TABLE is array (INTEGER range 1 .. 5) of FLOAT;
MY_LIST : TABLE := (3.7,  14.2,  -6.5,  0.0,  1.0);

type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);
type WEEK_ARRAY is array (DAYS) of BOOLEAN;

T : constant BOOLEAN := TRUE;
F : constant BOOLEAN := FALSE;
MY_WEEK : WEEK_ARRAY := (MON .. FRI => T, others => F);
```

MY_LIST

1	3.7
2	14.2
3	-6.5
4	0.0
5	1.0

MY_WEEK

SUN	FALSE
MON	TRUE
TUE	TRUE
WED	TRUE
THU	TRUE
FRI	TRUE
SAT	FALSE

```
MY_LIST (4) := 7.3;
if MY_WEEK (THU) = true then ...
if MY_WEEK (THU) then ...
```


MULTI-DIMENSIONED ARRAYS

```
subtype WEEKDAYS is DAYS range MON . . FRI;
type CLASS_PERIOD is range 1 . . 7;
type CLASSES is (HISTORY, ENGLISH, COMP_SCI, CALCULUS, FREE);
type SCHEDULE is array (WEEKDAYS,CLASS_PERIOD) of CLASSES;
MY_SCHEDULE : SCHEDULE;
```

	MON	TUE	WED	THU	FRI
1	CALCULUS	FREE	CALCULUS	FREE	CALCULUS
2	FREE	FREE	FREE	FREE	FREE
3	ENGLISH	FREE	ENGLISH	FREE	ENGLISH
4	COMP_SCI	COMP_SCI	COMP_SCI	COMP_SCI	COMP_SCI
5	FREE	FREE	FREE	FREE	FREE
6	FREE	FREE	FREE	FREE	FREE
7	FREE	FREE	FREE	FREE	FREE

```
if MY_SCHEDULE (WED, 3) = ENGLISH then . . .
```


ARRAYS OF ARRAYS

```
type DAY_SCHEDULE is array (CLASS_PERIOD) of CLASSES;
type WEEK_SCHEDULE is array (WEEKDAYS) of DAY_SCHEDULE;
MY_DAYS : WEEK_SCHEDULE;
```

	MON	TUE	WED	THU	FRI
1	CALCULUS	FREE	CALCULUS	FREE	CALCULUS
2	FREE	FREE	FREE	FREE	FREE
3	ENGLISH	FREE	ENGLISH	FREE	ENGLISH
4	COMP_SCI	COMP_SCI	COMP_SCI	COMP_SCI	COMP_SCI
5	FREE	FREE	FREE	FREE	FREE
6	FREE	FREE	FREE	FREE	FREE
7	FREE	FREE	FREE	FREE	FREE

```
if MY_DAYS (WED)(3) = ENGLISH then . . .
```


SLICES OF ONE-DIMENSIONAL ARRAYS

- A slice is a 'subarray'
- Slices have the same index type and component type as their parents
- A slice is created as an indivisible action, not component by component

```
type SLICE_EXAMPLE is array (1..7) of INTEGER;
MY_SLICE : SLICE_EXAMPLE := (1,2,3,4,5,6,7);
```

1	2	3	4	5	6	7
1	2	3	4	5	6	7

```
MY_SLICE (2 .. 4) := (8, 8, 8);
```

1	2	3	4	5	6	7
1	8	8	8	5	6	7

```
MY_SLICE (1 .. 4) := MY_SLICE (3 .. 6);
```

1	2	3	4	5	6	7
8	8	5	6	5	6	7

UNCONSTRAINED ARRAYS

- INDEX TYPE AND COMPONENT TYPE BOUND TO ARRAY TYPE
- INDEX RANGE BOUND TO OBJECTS, NOT TYPE
- ALLOWS FOR GENERAL PURPOSE SUBPROGRAMS
- INCLUDES Ada STRING TYPE

```
type SAMP is array (INTEGER range <>) of FLOAT;

LARGE : SAMP (1 .. 5) := (2.5, 3.4, 1.0, 0.0, 4.4);
SMALL : SAMP (2 .. 4)  := (2 .. 4 => 5.0);
```

LARGE

1	2.5
2	3.4
3	1.0
4	0.0
5	4.4

SMALL

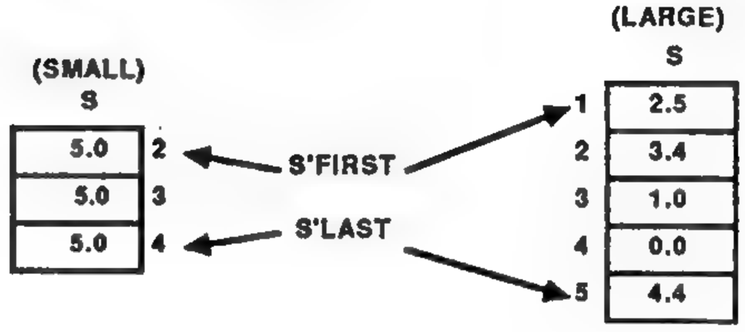
2	5.0
3	5.0
4	5.0

USING UNCONSTRAINED ARRAYS

```
function SUM (S : SAMP) return FLOAT is
   TOTAL : FLOAT := 0.0;
begin

   for INDEX in S'FIRST .. S'LAST
   loop
      TOTAL := TOTAL + S (INDEX);
   end loop;

   return TOTAL;

end SUM;
```

FUNCTION CALLS

```
put (SUM (SMALL));            -- 15.0

If SUM (LARGE) > 17.0 then ...   -- 11.3
```



Ada STRINGS

```
type STRING is array (POSITIVE range <>) of CHARACTER;  -- predefined

STR_5 : STRING (1 .. 5);

STR_6 : STRING (1 .. 6) := "Framus";
WARNING : constant STRING := "DANGER";

subtype TEN_LONG is STRING (1 .. 10);

FIRST_TEN : TEN_LONG := "HEADER     ";
```

STR_6

F	r	a	m	u	s
1	2	3	4	5	6

WARNING

D	A	N	G	E	R
1	2	3	4	5	6

FIRST_TEN

H	E	A	D	E	R				
1	2	3	4	5	6	7	8	9	10

CATENATION

- APPLIES TO ONE-DIMENSIONAL ARRAYS
- FOUR FORMS

LEFT	RIGHT	RESULT
ARRAY TYPE	ARRAY TYPE	ARRAY TYPE
ARRAY TYPE	COMPONENT TYPE	ARRAY TYPE
COMPONENT TYPE	ARRAY TYPE	ARRAY TYPE
COMPONENT TYPE	COMPONENT TYPE	ARRAY TYPE

```
STR : STRING ( 1 .. 9 ) := WARNING & ' ' &
                           STR_6 & " AHEAD";
```

Enter an appropriate range constraint

LOGICAL OPERATIONS ON BOOLEAN ARRAYS

- The logical operations of NOT, AND, OR and XOR are as appropriate for one-dimensional arrays whose component type is 'boolean' as they are for scalar objects of type 'boolean'

```
type BOOLS is array (1..4) of BOOLEAN;

T : constant BOOLEAN := TRUE;
F : constant BOOLEAN := FALSE;

P  : BOOLS := (T, T, F, F);
Q  : BOOLS := (T, F, T, F);
```

	P	Q	not P	P and Q	P or Q	P xor Q
1	T	T	F	T	T	F
2	T	F	F	F	T	T
3	F	T	T	F	T	T
4	F	F	T	F	F	F

ANONYMOUS ARRAY OBJECTS

```
A : array (1 .. 10) of BOOLEAN;
B : array (1 .. 10) of BOOLEAN;
```

- ANONYMOUS OBJECTS HAVE NO TYPE MARK
- CANNOT APPEAR AS RECORD COMPONENTS
- CANNOT BE PASSED AS PARAMETERS
- THE TWO ARRAYS ARE NOT COMPATIBLE

```
A, B : array (1 .. 10) of BOOLEAN;
```

- ARE THE TWO ARRAYS COMPATIBLE?

THE RANGE ATTRIBUTE

- APPLIES TO ALL ARRAY OBJECTS
- APPLIES TO ALL CONSTRAINED ARRAY TYPES
- DOES NOT APPLY TO ENUMERATION TYPES
- P'RANGE EQUATES TO P'FIRST .. P'LAST

```
type RANGE_EXAMPLE is array(1..4) of FLOAT;
SAMPLE : RANGE_EXAMPLE;
STR : STRING (1..10);
```

- THE FOLLOWING ARE VALID USES OF RANGE

```
RANGE_EXAMPLE'RANGE      -- 1 .. 4
SAMPLE'RANGE             -- 1 .. 4
STR'RANGE                -- 1 .. 10
```


NULL ARRAYS

- AN ARRAY WHICH CONTAINS NO COMPONENTS
- THE LOWER BOUND OF THE INDEX IS GREATER THAN THE UPPER BOUND
- ALLOWS THE 'EMPTY' STRING

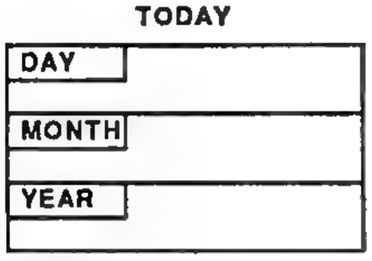
```
NULL_STRING : STRING(2 .. 1) := "";

. . .

for INDEX in NULL_STRING'RANGE
loop                          -- ignores the loop
  . . .
end loop;
```

RECORD TYPE DECLARATION

```
type DATE_TYPE is
   record
      DAY     : INTEGER range 1 .. 31;
      MONTH   : MONTH_TYPE;
      YEAR    : INTEGER range 1700 .. 2150;
   end record;
```

RECORD OBJECT DECLARATION

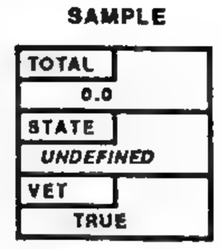
```
TODAY : DATE_TYPE;
```



DEFAULT RECORD COMPONENT VALUES

- If a component of a record type has a default value, every object declared to be of the record type will have that initial value.

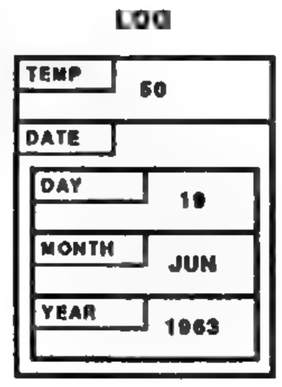
```
type DEFAULT_EXAMPLE is
   record
      TOTAL : FLOAT := 0.0;
      STATE : STATE_CODE;
      VET   : BOOLEAN := TRUE;
   end record;

SAMPLE : DEFAULT_EXAMPLE;
```



NESTED RECORDS

- COMPONENTS OF RECORDS MAY BE OF ANY TYPE, INCLUDING OTHER RECORDS
- THE VALUE OF A NESTED RECORD IS A NESTED AGGREGATE
- COMPONENT SELECTION USES EXTENDED 'DOTTED' NOTATION

```
type TEMPERATURE_LOG is
      record
            TEMP : INTEGER;
            DATE : DATE_TYPE;
      end record;

LOG : TEMPERATURE_LOG;
...
LOG.TEMP := 50;
LOG.DATE.DAY := 19;
LOG.DATE.MONTH := JUN;
LOG.DATE.YEAR := 1963;

-- or

LOG.DATE := (19, JUN, 1963);

-- or

LOG := (TEMP => 50,
        DATE => (19, JUN, 1963));

-- or

LOG := (50, (19, JUN, 1963));
```

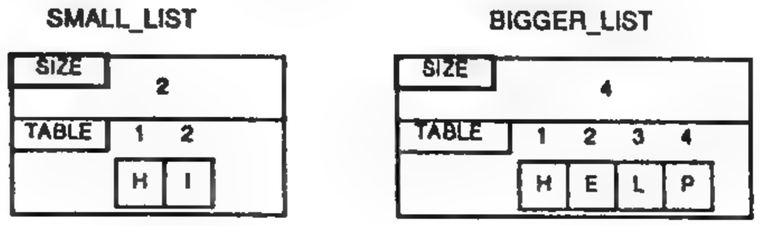


DISCRIMINATED RECORDS

- A discriminant is a special component of a record
- Discriminants must be of a discrete type
- Other components may depend on discriminants

```
subtype COUNTERS is INTEGER range 1 .. 100;

type MY_LIST (SIZE : COUNTERS) is
            record
                TABLE : STRING (1 .. SIZE);
            end record;

SMALL_LIST   : MY_LIST (SIZE => 2) := (2, ("HI"));

BIGGER_LIST : MY_LIST (4) := (4, ("HELP"));
```

DISCRIMINANT CONSTRAINT

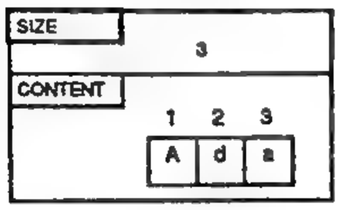


disc constrai
Object fixed for lifetim

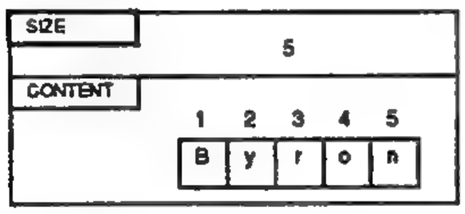
UNCONSTRAINED DISCRIMINATED RECORDS

If the discriminant has a default value and the object is declared using the default discriminant, then the discriminant can vary during execution.

```
type MSG_TYPE (SIZE : COUNTERS := 1) is
                record
                    CONTENT : STRING (1 .. SIZE);
                end record;

MESSAGE : MSG_TYPE;
...
MESSAGE := (3,"Ada");
```



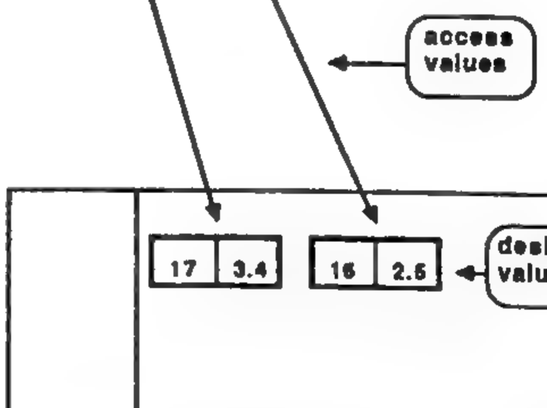

```
MESSAGE := (5, "Byron");
```



RECORD VARIANT PARTS

- Not only can array length be determined by a discriminant, but, the actual existance of certain fields can depend on a discriminant

```
type DEVICE is (PRINTER, DISK, DRUM);
type STATE  is (OPEN, CLOSED);

type PERIPHERAL (UNIT : DEVICE := DISK) is
      record
         STATUS : STATE;
         case UNIT is
            when PRINTER =>
                  LN_COUNT : NATURAL;
            when others =>
                  CYLINDER : NATURAL;
                  TRACK    : NATURAL;
         end case;
      end record;

WRITER  : PERIPHERAL (UNIT => PRINTER);
ARCHIVE : PERIPHERAL;
```



ACCESS TYPES

- DESIGNATED OBJECTS ARE DYNAMICALLY ALLOCATED (PERHAPS IN AN AREA OF A HEAP)
- ACCESS VALUES PROVIDE A WAY TO REFERENCE DESIGNATED OBJECTS
- ACCESS OBJECTS CONTAIN ACCESS VALUES AND ARE STATICALLY ALLOCATED (IN THE USER AREA) OR APPEAR IN DESIGNATED OBJECTS (AS LINKS)

```
type SAMPLE is
    record
       AGE : NATURAL;
       GPA : FLOAT;
    end record;
```

ACCESS TYPE

```
type PTR is access SAMPLE;
```

ACCESS OBJECTS

```
JOHN, MARY : PTR;
```


ALLOCATORS

```
MARY := new SAMPLE '(AGE => 16, GPA => 2.5);
JOHN := new SAMPLE '(17, 3.4);
```

Storage_error — no heap space

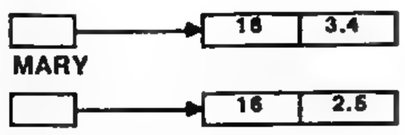
numeric_error

constraint_error

DEREFERENCING

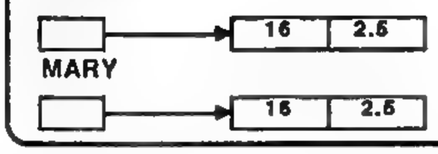
```
JOHN.AGE := MARY.AGE;        -- component assignment
```

JOHN



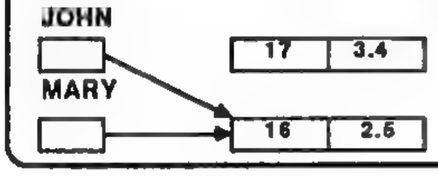

```
JOHN.all := MARY.all;        -- entire object assignment
```

JOHN




```
JOHN := MARY;        -- access value assignment
```

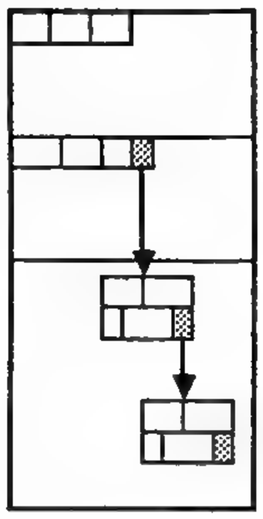
JOHN



ACCESS TYPES

(Memory Allocation)

```
type NODE;  -- incomplete type decl.

type PTR is access NODE;

type NODE is
     record
          FIELD_1 : SOME_TYPE;
          FIELD_2 : BLAH;
          FIELD_3 : FOO;
          FIELD_4 : FRAMUS;
          FIELD_5 : PTR;
     end record;
```




```
TOP : PTR;  -- an access object
...
TOP := new NODE;  -- an allocator

TOP.FIELD_5 := new  NODE;  -- another allocator
```


PRIVATE TYPES

- ACTUAL TYPE DESCRIPTION IS 'HIDDEN'
- THE TYPE IS PRIMARILY KNOWN THRU ITS OPERATIONS
- PRIVATE TYPES ARE ALWAYS IMPLEMENTED BY PACKAGES
- PRIVATE TYPES PROTECT DATA FROM ERRONEOUS ACCESS
- IF AN OBJECT IS OF A PRIVATE TYPE, ASSIGNMENT, (IN)EQUALITY AND ALL EXPLICITLY DECLARED OPERATIONS ARE ALLOWED
- IF AN OBJECT IS OF A LIMITED PRIVATE TYPE , ONLY THE EXPLICITLY DECLARED OPERATIONS ARE ALLOWED

DERIVED TYPES

- INHERITS ALL VALUES (WITH AN OPTIONAL CONSTRAINT) AND ALL OPERATIONS (INCLUDING USER-DEFINED) FROM A PARENT TYPE
- THE DERIVED TYPE AND THE PARENT TYPE ARE NOT IMPLICITLY COMPATIBLE
- TYPE TRANSFER BETWEEN PARENT AND DERIVED TYPE IS PERMITTED
- TYPE TRANSFER BETWEEN OBJECTS OF TWO DIFFERENT TYPES DERIVED FROM THE SAME PARENT IS PERMITTED

```
type MY_STRING_TYPE is new STRING;
MY_STRING : MY_STRING_TYPE (1 .. 10);
STR : STRING (1 .. 10);
...
MY_STRING := MY_STRING_TYPE (STR);
```

TYPE TRANSFER

SUBPROGRAM DECLARATIONS

```
procedure GENERATE_HEADING;

procedure PUSH ( E    : in ELEMENT;
                 ON   : in out STACK);

procedure INCREMENT (COUNT : in out COUNTER);

function SQRT (ARG : FLOAT) return FLOAT;

function GET_NEXT return CHARACTER;

function "+" (S1, S2 : SET) return SET;

function INVERT (S : STRING) return STRING;
```

- DEFAULT PARAMETERS (IN)

```
function FIND ( SUB_STRING  : STRING;
                TARGET      : STRING;
                START       : INTEGER := 1)
                              return INTEGER;
```


SUBPROGRAM CALLS

```
GENERATE_HEADING;

PUSH (NEW_ELEMENT, ON => MY_STACK);

INCREMENT (TALLY);

STD_DEV := SQRT (VARIANCE);

LETTER := GET_NEXT;

SET_OF_PETS := SET_OF_CATS + SET_OF_DOGS;

PALINDROME := INVERT(S) = S;

MY_INDEX := FIND ("Hello",  MESSAGE, START => 5);

MY_INDEX := FIND ("Hello", MESSAGE);
```


OVERLOADING

- THE DESIGNATORS (IDENTIFIER OR SYMBOL) OF SUBPROGRAMS NEED NOT BE UNIQUE
- AMBIGUITY CAN BE RESOLVED BY COMPARING PARAMETER AND RESULT TYPE PROFILES OR BY QUALIFICATION
- TYPE PROFILES

  -- NUMBER OF PARAMETERS
  -- TYPES OF PARAMETERS (BY POSITION)
  -- TYPE OF RESULT (FUNCTIONS ONLY)

- AMBIGUITIES WILL BE REPORTED BY THE COMPILER

OVERLOAD RESOLUTION

```
type COLOR is (RED, GREEN, BLUE, ORANGE);
type LIGHT  is (RED, YELLOW, GREEN);

procedure SET (HUE : COLOR);
procedure SET (HUE : LIGHT);
procedure SET (SPOT : INTEGER);
procedure SET (FLAG : BOOLEAN);
```

CALLS

```
SET (BLUE);
SET (17);
SET (TRUE);
SET (RED);                  -- ambiguous
SET (LIGHT'(RED));
```

QUALIFICATION

SUBPROGRAM BODIES

```
<subprogram_body> ::=
    <subprogram_specification> is
        [<declarative_part>]
    begin
        <sequence_of_statements>
    [exception
        <exception_handler>
        {<exception_handler>}]
    end [<designator>];
```


BLOCK STRUCTURE

SUBPROGRAMS CAN BE NESTED

AN OBJECT DECLARED IN A BLOCK AND REFERENCED IN THE SAME BLOCK IS SAID TO BE 'LOCAL' TO THAT BLOCK

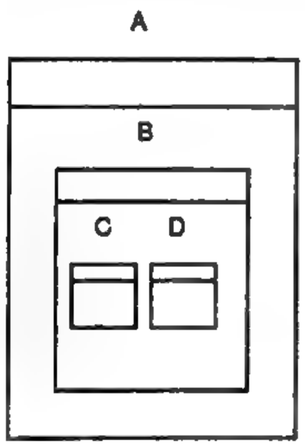
AN OBJECT DECLARED IN A BLOCK AND REFERENCED IN AN INNER BLOCK IS SAID TO BE 'GLOBAL' TO THAT INNER BLOCK

AN OBJECT DECLARED IN AN INNER BLOCK IS INACCESSIBLE FROM AN OUTER BLOCK

AN OBJECT DECLARED IN AN INNER BLOCK CAN BE A HOMOGRAPH OF AN OBJECT DECLARED IN AN OUTER BLOCK AND WILL 'HIDE' THE OBJECT IN THE OUTER BLOCK

A MODEL OF SUBPROGRAM ACTIVATION

STATIC VIEW



DYNAMIC VIEW

A is called
A calls B
B calls C
C calls D
D calls C
C returns
D returns
C returns
B returns
A returns

Consider the implementation of a subprogram to be in two parts: a unique code segment and an activation record

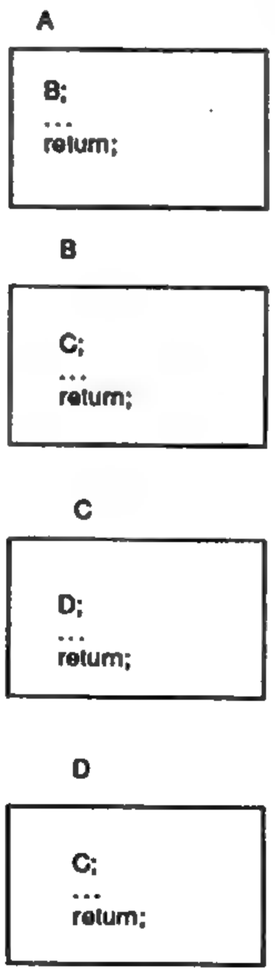
code segment

reentrant code, no data maintained

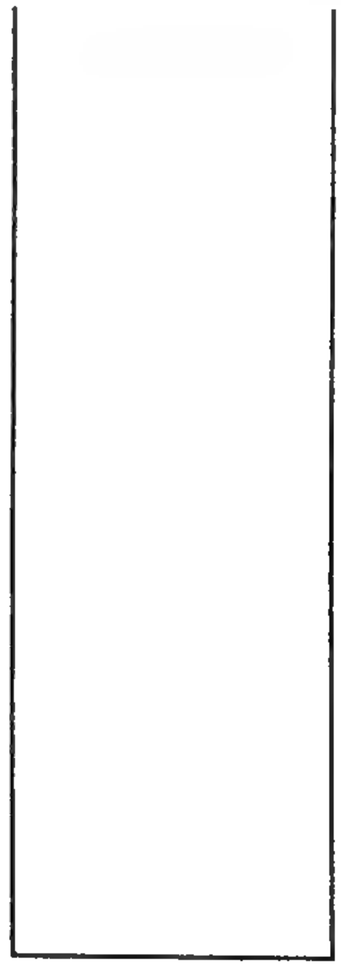
activation record

parameters	
local variables	
return point	
static link	dynamic link

code segments



run-time stack



ASSIGNMENT STATEMENTS

- The variable takes on the value of the expression
- The variable and the expression must be of the same type

```
MY_INT    := 17;                  -- integer
LIST(2..4) := LIST (7 .. 9);      -- slice
TODAY     := (13, DEC, 1964);     -- aggregate
X         := SQRT (Y);            -- function call
```


NULL STATEMENT

- Used when no action is to take place
- Explicit 'null' avoids problem which arise in some languages by using the 'empty' statement

```
case FRAMUS is
  when 1      => <seq-of-stmts>
  when 2      => <seq-of-stmts>
  when others => null;
end case;
```


PROCEDURE CALL

- A procedure call is a sequential statement in an 'extended' language
- A well-named procedure exemplifies both abstraction and information hiding

```
DISPLAY (TODAYS_DATE);
RAISE_ALARM;
TEXT_IO.PUT_LINE (THE_MESSAGE);
```


RETURN STATEMENT

- RETURN STATEMENTS ONLY OCCUR IN SUBPROGRAMS
- WHEN A RETURN STATEMENT IS EXECUTED, CONTROL IS IMMEDIATELY PASSED TO THE POINT OF CALL
- 'RETURNS' FROM FUNCTIONS MUST BE ASSOCIATED WITH AN EXPRESSION
- 'RETURNS' FROM PROCEDURES ARE ALTERNATIVES TO 'FALLING THROUGH THE BOTTOM' OF THE PROCEDURE

```
procedure DO_IT is
 begin
   if ... then
      <stmt>
      <stmt>
      return;
   end if;
      <stmt>
      <stmt>
end DO_IT;
```

fall through/under
program_error

RETURN FROM FUNCTION

- Falling through the bottom of a function results in a PROGRAM_ERROR exception

```
function SQRT (ARG : FLOAT) return   FLOAT   is

  RESULT : FLOAT;

begin
   -- statements to calculate RESULT

      return   RESULT;

exception
   -- either a RAISE or RETURN statement must appear here

end SQRT;
```


BLOCK STATEMENTS

- A block statement provides localization for

  -- declarations
  -- exceptions
  -- or both

```
declare
   TEMP : INTEGER := X;
begin
   X := Y;
   Y := TEMP;
end;
```

```
begin
    GET (MY_VALUE);
exception
   when CONSTRAINT_ERROR =>
        -- action for dealing with error
end;
```


BLOCK STATEMENTS

```
...
EXAMPLE:
  declare

     I : INTEGER;
     procedure SUB is . . .

  begin

       INT_IO.GET (I);
       SUB;

  exception

    when NUMERIC_ERROR | CONSTRAINT_ERROR =>
       DO_SOMETHING;

  end EXAMPLE;
...
```

The name EXAMPLE.I is available within the procedure SUB.

BLOCK STATEMENTS

```
<block_statement> ::=
  [<block_simple_name>:]
    [declare

        <declarative_part>]

    begin

        <sequence_of_statements>

    [exception

        <exception_handler>
        {<exception_handler>}]

    end [<block_simple_name>];
```

After exceptions are handled, control passes to the next sequential instruction

CONDITIONAL STATEMENTS (IF)

```
if TODAY.DAY = 30 and TODAY.MONTH = JUL then
   PEGS_YEARS := PEGS_YEARS + 1;
   GET (BIRTHDAY_CARD);
end if;


if IS_ODD (NUMBER) then
   ODD_TOTAL := ODD_TOTAL + 1;
else
   EVEN_TOTAL := EVEN_TOTAL +1;
end if;


if      SCORE >= 90 THEN  GRADE := 'A';
elsif   SCORE >= 80 THEN  GRADE := 'B';
elsif   SCORE >= 70 THEN  GRADE := 'C';
elsif   SCORE >= 60 THEN  GRADE := 'D';
else                      GRADE := 'E';
end if;
```


CONDITIONAL STATEMENTS (IF)

```
<if_statement> ::=

    if <condition> then
        <sequence_of_statements>
    {elsif <condition> then
        <sequence_of_statements>}
    [else
        <sequence_of_statements>]
    end if;
```


CONDITIONAL STATEMENTS (CASE)

```
procedure SWITCH (HEADING : in out DIRECTION) is

  begin

    case HEADING is

        when NORTH  => HEADING := SOUTH;
        when EAST   => HEADING := WEST;
        when SOUTH  => HEADING := NORTH;
        when WEST   => HEADING := EAST;

    end case;

  end SWITCH;


case NUMBER is

    when 2          => <sequence_of_statements>
    when 3 | 7 | 8  => <sequence_of_statements>
    when 9 .. 20    => <sequence_of_statements>
    when others     => <sequence_of_statements>

end case;
```


CONDITIONAL STATEMENTS (CASE)

```
<case statement> ::=

    case  <discrete_expression> is
        when <choice> {|<choice>}  =>
            <sequence_of_statements>
        {when <choice> {|<choice>}  =>
            <sequence_of_statements> }
    end case;


<choice> ::=

        <discrete_expression>|
        <discrete_range>|
        others
```

NOTE: THE CHOICES MUST BE MUTUALLY EXCLUSIVE (NO VALUE IS TREATED MORE THAN ONCE) AND ALSO COLLECTIVELY EXHAUSTIVE (EVERY VALUE IS TREATED)

ITERATION STATEMENTS

```
loop
   GET (SAMPLES);
   PROCESS (SAMPLES);
end loop;
```

```
loop
   GET (NUMBER);
   exit when NUMBER = 0;
   PROCESS (NUMBER);
end loop;
```

```
while DATA_REMAINS
loop
   <sequence_of_statements>
end loop;
```

```
OUTER:
  loop
     <sequence_of_statements>

     loop
        <sequence_of_statements>
        exit OUTER when NUMBER > 7;
     end loop;

     <sequence_of_statements>
end loop OUTER;
```


CONTROL VARIABLES

- ARE IMPLICITLY DECLARED
- MUST BE DISCRETE
- TAKE THEIR TYPE FROM THE DISCRETE RANGE
- ARE IN EXISTENCE ONLY UNTIL end loop
- CAN 'HIDE' A VARIABLE WITH SAME NAME
- CANNOT BE MODIFIED (LOCAL CONSTANT)
- ONLY SINGLE STEP INCREMENT (DECREMENT)

```
for INDEX in DAYS    -- SUN .. SAT
loop
   ...
end loop;


for COUNTER in reverse 1 .. 10
loop
   ...
end loop;
```


AN ADA IDIOM FOR INPUT

Input of numeric data can generate exceptions:

```
INT_IO.GET(MY_NUMBER);
```

This statement could, when expecting data from the keyboard, receive characters which do not conform to the syntax of the base type of MY_NUMBER. An exception handler is, therefore, appropriate.

```
        INT_IO.GET(MY_NUMBER);

exception

        when TEXT_IO.DATA_ERROR =>
           ...
```

But, exception handlers can occur only in block statements or in bodies of subprograms, packages and tasks. We shall use a block statement to achieve our purpose.

```
...
begin   -- block statement

        INT_IO.GET(MY_NUMBER);

exception

        when TEXT_IO.DATA_ERROR =>
           <sequence_of_statements>

end;    -- block statement

...
```


AN ADA IDIOM FOR INPUT

But, we probably want the user to be able to repeat the action until no error occurs. Therefore, we encase the block statement inside of a loop statement.

```
...
loop

   begin

        INT_IO.GET(MY_NUMBER);

   exception

        when TEXT_IO.DATA_ERROR => ...

   end;

end loop;
...
```

But, this allows us no way to leave the loop. Therefore, we complete the idiom by inserting an exit statement which will be executed only if the INT_IO.GET statement does not raise an exception.

```
...
loop
   begin
        INT_IO.GET(MY_NUMBER);
        exit;
   exception
        when TEXT_IO.DATA_ERROR =>
             <sequence_of_statements>
   end;
end loop;
...
```

Write a program which will take the frequency count of the letters in a string (message). The user of the program should be able to indicate how many elements of the freq count are to be printed per line.

TEXT: "AMWAY FOLKS WRITE COBOL IN ADA"

COLUMNS : 4

OUTPUT:

A = 4	B = 1	C = 1	D = 1
E = 1	F = 1	G = 0	H = 0
I = 2	J = 0	K = 1	L = 2
M = 1	N = 1	O = 3	P = 0
Q = 0	R = 1	S = 1	T = 1
U = 0	V = 0	W = 2	X = 0
Y = 1	Z = 0		

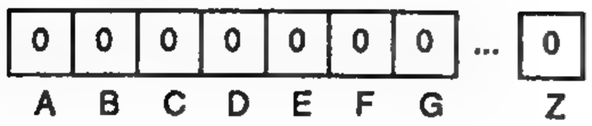
array = type
Message first = 1


```
with TEXT_IO;
procedure MAIN is

  subtype ALPHA is CHARACTER range 'A' .. 'Z';
  type FREQ_TABLE is array (ALPHA) of NATURAL;

  COLUMNS : NATURAL;
  package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

  function FREQ (MSG : STRING)
                    return FREQ_TABLE is separate;

  procedure PRINT (TABLE : FREQ_TABLE;
                   UNITS_PER_LINE : NATURAL) is separate;

begin

  TEXT_IO.PUT_LINE ("How many columns of output " &
                                  "per line? (enter 1 to 10)");

  INT_IO.GET (COLUMNS);

  PRINT (TABLE => FREQ ("AMWAY FOLKS WRITE" &
                        " COBOL IN ADA"),
         UNITS_PER_LINE => COLUMNS);

end MAIN;
```



```
separate (MAIN)
function FREQ (MSG : string) return FREQ_TABLE is

    TABLE : FREQ_TABLE := ('A' .. 'Z' => 0);

begin

    for INDEX in MSG'range
    loop

        if MSG(INDEX) in ALPHA then

            TABLE (MSG (INDEX)) :=
                TABLE (MSG (INDEX)) + 1;

                -- TABLE ('A') := TABLE ('A') + 1;  etc.

        end if;

    end loop;

    return TABLE;

end FREQ;
```

TABLE



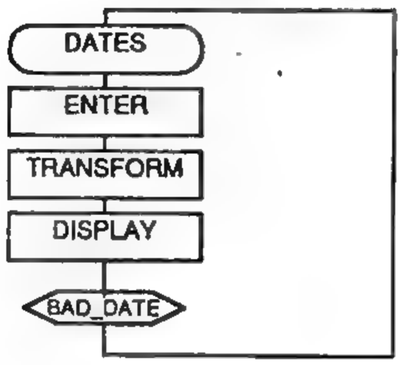

```
separate (MAIN)
procedure PRINT (TABLE          : FREQ_TABLE;
                 UNITS_PER_LINE : NATURAL)  is

    CH : ALPHA := ALPHA'FIRST;   -- 'A'

begin

    OUTER:       -- a named loop
    loop
      for I in 1 .. UNITS_PER_LINE
      loop

                          -- output 1 element
          TEXT_IO.PUT (CH);
          TEXT_IO.PUT (" = ");
          INT_IO.PUT (TABLE(CH));
          TEXT_IO.PUT ("    ");

          exit OUTER when CH = ALPHA'LAST;
          CH := ALPHA'SUCC (CH);
      end loop;  -- for I

      TEXT_IO.NEW_LINE;


    end loop OUTER;      -- only when 'Z'

    TEXT_IO.NEW_LINE;

end PRINT;
```

GENERATE TOMORROW'S DATE

The user enters today's date and the date is displayed. The date is transformed into tomorrow's date and the new date is displayed. Invalid dates raise exceptions.

The primary objects of interest are DATES. Operations on DATES are: ENTER, DISPLAY, and TRANSFORM. A bad date (such as 30 FEB) should raise an exception.

DATE_PACKAGE



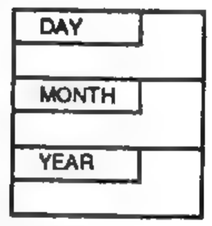
The Ada package specification

```
package DATE_PACKAGE is

    type DATES is private;
    procedure ENTER     (D : out DATES);
    procedure DISPLAY   (D : in DATES);
    function  TRANSFORM (D : DATES) return DATES;
    BAD_DATE : exception;

private
    ...

end DATE_PACKAGE;
```



```
with DATE_PACKAGE, TEXT_IO;

procedure CHANGE_DATE is

    TODAY, TOMORROW : DATE_PACKAGE.DATES;

begin

    DATE_PACKAGE.ENTER (TODAY);
    TEXT_IO.PUT ("Today is . . .");
    DATE_PACKAGE.DISPLAY (TODAY);
    TOMORROW := DATE_PACKAGE.TRANSFORM (TODAY);
    TEXT_IO.PUT (" and tomorrow is . . .");
    DATE_PACKAGE.DISPLAY (TOMORROW);

exception

    when DATE_PACKAGE.BAD_DATE =>
        TEXT_IO.PUT_LINE ("Invalid date, restart process.");

end CHANGE_DATE;
```


The complete Ada package specification




```
package DATE_PACKAGE is

    type DATES is private;
    procedure ENTER     (D : out DATES);
    procedure DISPLAY   (D : in DATES);
    function  TRANSFORM (D : DATES) return DATES;
    BAD_DATE : exception;

private

    type MONTH_TYPE is ( JAN, FEB, MAR, APR, MAY, JUN,
                         JUL, AUG, SEP, OCT, NOV, DEC);

    type DATES is
        record
            DAY   : NATURAL range 1 .. 31;
            MONTH : MONTH_TYPE;
            YEAR  : NATURAL range 1800 .. 2150;
        end record;

end DATE_PACKAGE;
```

```
with TEXT_IO;
package body DATE_PACKAGE is

    package MONTH_IO is new TEXT_IO.ENUMERATION_IO (MONTH_TYPE);
    package INT_IO is new TEXT_IO.INTEGER_IO (NATURAL);

   -- bodies of all subprograms go here

end DATE_PACKAGE;
```

```
procedure DISPLAY (D : in DATES) is

begin

      MONTH_IO.PUT(D.MONTH);

      INT_IO.PUT (D.DAY, 3);

      TEXT_IO.PUT (',');

      INT_IO.PUT (D.YEAR, 5);

end DISPLAY;
```

D

DAY
MONTH
YEAR

D

DAY
MONTH
YEAR

```
procedure ENTER (D : out DATES) is

   type DATE_PROMPTS is (DD, MM, YY);

begin

   for SELECTOR in DATE_PROMPTS
      loop   -- outer loop for stepped iteration

         loop   -- inner loop to contain block
            begin   -- local block to contain exception handler
              case SELECTOR is
                 when DD => TEXT_IO.PUT_LINE ("day:");
                            INT_IO.GET (D.DAY);
                 when MM=> TEXT_IO.PUT_LINE ("month:");
                            MONTH_IO.GET (D.MONTH);
                 when YY => TEXT_IO.PUT_LINE ("year:");
                            INT_IO.GET (D.YEAR);
              end case;
              exit;          -- leave the inner-most loop
            exception
              when TEXT_IO.DATA_ERROR | CONSTRAINT_ERROR =>
                case SELECTOR is
                  when DD =>
                     TEXT_IO.PUT_LINE ("enter integer 1 to 31");
                  when MM =>
                     TEXT_IO.PUT_LINE ("enter 3-ltr month");
                  when YY =>
                     TEXT_IO.PUT_LINE ("enter 4-digit year");
                end case;
            end;   -- local block
         end loop;   -- inner loop containing block

      end loop;   --outer loop controlling iteration

end ENTER;
```

proper body - textual nesting
- stub - separate


```
function DAYS_IN_MONTH (D : DATES) return NATURAL is

begin

    case D.MONTH is

      when SEP | APR | JUN | NOV => return 30;

      when FEB =>
         if ((D.YEAR mod 4 = 0) and (D.YEAR mod 100 /= 0))
                or
            (D.YEAR mod 400 = 0) then
            return 29;
         else
            return 28;
         end if;

      when others => return 31;

    end case;

end DAYS_IN_MONTH;
```

D

DAY
MONTH
YEAR


```
function TRANSFORM (D : DATES) return DATES is

   LAST_DAY : constant NATURAL := DAYS_IN_MONTH (D);

begin

   If D.DAY > LAST_DAY then
      raise BAD_DATE;
   end if;

   If D.DAY /= LAST_DAY then
      return (D.DAY + 1, D.MONTH, D.YEAR);
   end if;

   If D.MONTH /= MONTH_TYPE'LAST then
      return (1, MONTH_TYPE'SUCC (D.MONTH), D.YEAR);
   end if;

   return (1, MONTH_TYPE'FIRST, D.YEAR + 1);
end TRANSFORM;
```

D

DAY
MONTH
YEAR

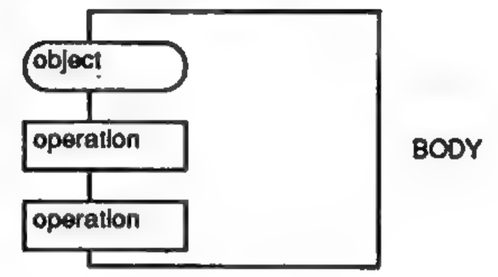
It was clear that a most powerful addition to any programming language would be the ability to define new higher level entities in terms of previously known ones, and then to call them by name. This would build the chunking right into the language. Instead of there being a determinate repertoire of instructions out of which all programs had to be explicitly assembled, the programmer could construct his own modules, each with its own name, each usable anywhere inside the program, just as if it had been a built-in feature of the language.

-- Douglas Hofstadter
"Goedel, Escher, Bach"

Package Specification -- the contract

```
<package specification> ::=
        package <identifier> is
            {<basic_declarative_item>}
        [private
            {<basic_declarative_item>}]
        end [<package_simple_name>];
```

SPECIFICATION (visible)



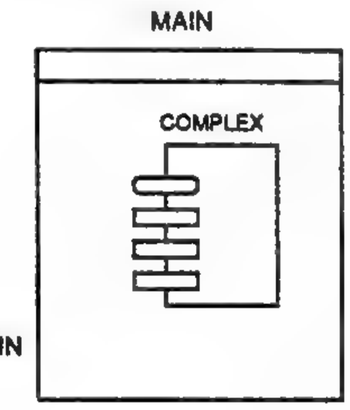
PACKAGE VISIBILITY

- A PACKAGE CAN BE MADE AVAILABLE IN TWO DISTINCT WAYS
  - -- It can be textually nested (rarely used)
  - -- It can be accessed from a library

```
package COMPLEX is

    type NUMBER is
      record
        REAL_PART : FLOAT;
        IMAGINARY_PART : FLOAT;
      end record;

    function "+" (X,Y : NUMBER) return NUMBER;
    function "-"  (X,Y : NUMBER) return NUMBER;
    function "*"  (X,Y : NUMBER) return NUMBER;

end COMPLEX;
```

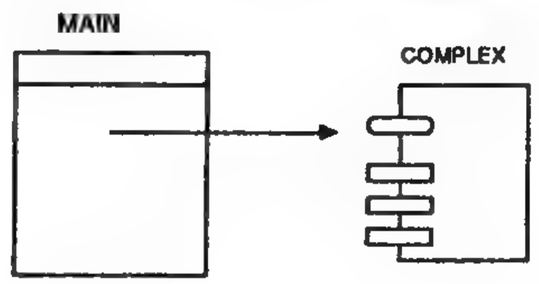

TEXTUALLY NESTED PACKAGES

```
procedure MAIN is
   ...
   package COMPLEX is
      type NUMBER...
      function "+"...
      function "-"...
      function "*"...
   end COMPLEX;
   ...
   package body COMPLEX is...
   ...
begin
      -- sequence of statements for MAIN
end MAIN;
```



PACKAGES AS LIBRARY UNITS

```
with COMPLEX;
procedure MAIN is...
```



PACKAGE SPECIFICATIONS

- A package specification contains only basic declarative items (no bodies allowed)
- The user 'imports' the package resources
- The package 'exports' the resources
- The 'with' clause gives the user visibility of the package resources (dotted notation must be used)
- The 'use' clause gives the user direct visibility of the package resources (simple names can be used)

```
with COMPLEX; use COMPLEX;
procedure SAMPLE is
    NUMBER_1, NUMBER_2 : NUMBER;
begin
    ...
    NUMBER_1 := NUMBER_1 * NUMBER_2;
    ...
end SAMPLE;
```


PACKAGE BODIES -- THE IMPLEMENTATION

```
<package body> ::=
    package body <package_simple_name> is
      [<declarative_part>]
    [begin
       <sequence_of_statements>
    [exception
       <exception_handler>
      {<exception_handler>}]]
     end [<package_simple_name>];
```


PACKAGE BODIES

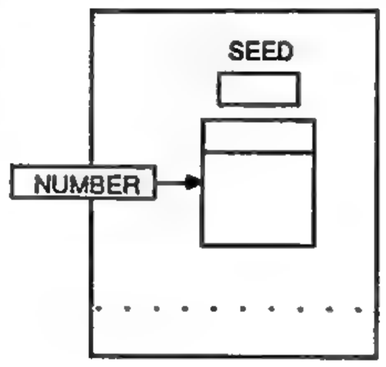
- IF A UNIT (subprogram, package, task, generic) SPECIFICATION OCCURS IN THE PACKAGE SPECIFICATION THEN THE UNIT BODY MUST OCCUR IN THE PACKAGE BODY.
- IF THERE ARE NO SUCH UNIT SPECIFICATIONS IN THE PACKAGE SPECIFICATION, THE PACKAGE BODY IS OPTIONAL.
- THE OPTIONAL SEQUENCE OF STATEMENTS IN THE PACKAGE BODY IS EXECUTED ONE TIME WHEN THE PACKAGE IS ELABORATED.
- IF THE PACKAGE IS TEXTUALLY NESTED IN THE DECLARATIVE PART OF SOME OTHER UNIT, THEN THE BODY OF THE PACKAGE CAN BE NESTED AS A BODY STUB AND THE PROPER BODY CAN BE COMPILED SEPARATELY AS A SUBUNIT.

PACKAGE BODIES

```
package body COMPLEX is

  function "+" (X,Y : NUMBER) return NUMBER is

    RESULT : NUMBER;

  begin
    RESULT.REAL_PART :=
        X.REAL_PART + Y.REAL_PART;

    RESULT.IMAGINARY_PART :=
        X.IMAGINARY_PART + Y.IMAGINARY_PART;

    return RESULT;
  end "+";

  function "-" (X,Y : NUMBER) return NUMBER is
  begin
     return  (REAL_PART =>
                  X.REAL_PART - Y.REAL_PART,
              IMAGINARY_PART =>
                  X.IMAGINARY_PART - Y.IMAGINARY_PART);
  end "-";

  function "*" . . .

end COMPLEX;
```

BODIES WITH A BLOCK STATEMENT

```
package RANDOM is
   function NUMBER return FLOAT;
end RANDOM;

with TEXT_IO;
package body RANDOM is

   SEED : INTEGER;

   package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);

   function NUMBER return FLOAT is
      ...
   end NUMBER;

begin

   TEXT_IO.PUT_LINE ("enter 5-digit odd number:");
   INT_IO.GET(SEED);
   -
   -- error checking routine
   -
end RANDOM;
```



PRIVATE TYPES

- THE USER OF A LIMITED PRIVATE TYPE CAN ONLY USE THE PROVIDED (EXPORTED) OPERATIONS
- THE USER OF A PRIVATE TYPE CAN, ADDITIONALLY USE THE (IN)EQUALITY AND ASSIGNMENT OPERATIONS
- THE IMPLEMENTOR OF THE PRIVATE TYPE HAS NO SUCH RESTRICTIONS WHEN WRITING THE PACKAGE BODY
- THE FOLLOWING BASIC OPERATIONS ARE ALSO NOT ALLOWED WHEN USING PRIVATE TYPES:

1. Dynamic allocation
2. Test for membership
3. A short-circuit control form
4. Component selection
5. Compnent indexing
6. Slice
7. Qualification
8. Type conversion
9. Literals
10. Aggregates
11. Attributes


```
package INT_STACK_INFO is

   type STACK is limited private;

   procedure PUSH (ITEM : in INTEGER;
                   ON   : in out STACK);

   procedure POP  (ITEM : out INTEGER;
                   FROM : in out STACK);

   EMPTY_STACK,
   FULL_STACK     : exception;

private

   type STACK is ...

end INT_STACK_INFO;
```


NAMED COLLECTION OF DECLARATIONS

- Package body is optional

```
package DATE_INFO is

   type DAY_NAME is ( MON, TUE, WED, THU,
                      FRI, SAT, SUN);

   type DAY_VALUE is range 1 .. 31;

   type MONTH_NAME is ( JAN, FEB, MAR, APR,
                        MAY, JUN, JUL, AUG,
                        SEP, OCT, NOV, DEC);

   type YEAR_VALUE is range 0 .. INTEGER'LAST;

   type DATE_TYPE is
      record
         DAY   : DAY_VALUE;
         MONTH : MONTH_NAME;
         YEAR  : YEAR_VALUE;
      end record;

end DATE_INFO;
```

ABSTRACT STATE MACHINE

- USED WHEN THERE IS ONLY ONE OBJECT OF A GIVEN TYPE
- MAINTAINS 'KNOWLEDGE' OF THAT OBJECT WITHIN THE PACKAGE BODY
- ELIMINATES NEED TO PASS OBJECT BACK AND FORTH VIA PARAMETERS

```
package FURNACE is
   function IS_RUNNING return BOOLEAN;
   procedure SET (TEMP : in FLOAT);
   procedure SHUT_DOWN;
   function TEMP_IS return FLOAT;
   OVERTEMP : exception;
end FURNACE;
```


ABSTRACT DATA TYPE

- USED WHEN THERE ARE MORE THAN ONE OBJECT OF A GIVEN TYPE
- NO INFORMATION ABOUT THE INDIVIDUAL OBJECT IS MAINTAINED IN THE PACKAGE BODY
- OBJECTS ARE DECLARED IN THE USING UNIT AND ARE PASSED BACK AND FORTH VIA PARAMETERS.

```
package FURNACE_STUFF is
   type FURNACE is . . .
   function IS_RUNNING (F : FURNACE) return BOOLEAN;
   procedure SET (F: in out FURNACE; TEMP : in FLOAT);
   procedure SHUT_DOWN (F : in FURNACE);
   function TEMP_IS (F : FURNACE) return FLOAT;
   OVERTEMP : exception;
end FURNACE_STUFF;
```


OPERATIONS ON OBJECTS

- CONSTRUCTORS

  -- ALTER THE VALUE OF AN OBJECT
  -- USUALLY A PROCEDURE

- SELECTORS

  -- RETURN THE VALUE OF AN OBJECT
  -- USUALLY A FUNCTION

- ITERATORS

  -- PROVIDE A MECHANISM TO VISIT ALL OBJECTS
  -- IMPLEMENTED AS A PRIVATE TYPE AND

    --- A MEANS OF INITIALIZATION THE ITERATOR
    --- A MEANS OF RETRIEVING AN OBJECT
    --- A MEANS OF INCREMENTING THE ITERATOR
    --- A MEANS OF DETERMINING COMPLETION

QUEUE PACKAGE

```
package QUEUE_OF_INTEGERS is

   type QUEUE is private;
   function MAKE return QUEUE;
   procedure ADD (INT :  in INTEGER; TO : in out QUEUE);
   procedure REMOVE (INT : out INTEGER; FROM : in out QUEUE);
   function SIZE_OF (Q : QUEUE) return NATURAL;

   procedure INITIALIZE_ITERATION;
   function NEXT_VALUE_OF_ITERATION  return INTEGER;
   function ITERATION_IS_COMPLETE return BOOLEAN;

   QUEUE_FULL, QUEUE_EMPTY, ITERATION_ERROR : exception;
private
   type QUEUE is . . .
end QUEUE_OF_INTEGERS;
```

```
package BOUNDED_LENGTH_STRING is

    type TEXT is private;

    MAX_SIZE : constant := 1000;

    type SIZE is range 0 .. MAX_SIZE;

    type INDEX is range 0 .. MAX_SIZE;
       -- an INDEX of 0 reflects a failed search

    INDEX_ERROR, SIZE_ERROR : exception;
    ------------------------------------------------------------------------
    procedure INSERT
            (SUB_TEXT : TEXT; ORIGINAL : in out TEXT; START : INDEX);
    procedure INSERT
            (SUB_TEXT : STRING; ORIGINAL : in out TEXT; START : INDEX);
          -- The SUB_TEXT is inserted into the ORIGINAL text beginning
          -- at START.  SIZE_ERROR or INDEX_ERROR can occur.

    ------------------------------------------------------------------------
    procedure DELETE
              (ORIGINAL : in out TEXT; START : INDEX; COUNT : SIZE);

          -- COUNT characters are removed from the ORIGINAL text
          -- beginning at START.  SIZE_ERROR or INDEX_ERROR
          -- can occur.

    ------------------------------------------------------------------------
    function "&" (HEAD : TEXT; TAIL : TEXT) return TEXT;

          -- the TAIL is catenated to the back of the HEAD.
          -- SIZE_ERROR can occur.

    ------------------------------------------------------------------------
    function COPY (SOURCE : TEXT; START : INDEX; COUNT : SIZE)
                                                  return TEXT;

          -- Returns text composed by selecting COUNT characters
          -- from the SOURCE text beginning at index START.
          -- INDEX_ERROR or SIZE_ERROR can occur
```


Read in a string of TEXT. Replace the first occurrence (if any) of "FRAMUS" BY "PHONORTON". Print out the resulting TEXT. Treat any exceptions which might arise.


```
    function LENGTH (SOURCE : TEXT) return SIZE;

          -- Returns current size of the SOURCE.
    ------------------------------------------------------------------------
    function POS (  PATTERN : TEXT;
                    SOURCE  : TEXT;
                    START   : INDEX := 1) return INDEX;

    function POS (  PATTERN : STRING;
                    SOURCE  : TEXT;
                    START   : INDEX := 1) return INDEX;

          -- Returns the beginning location of the first occurrence of PATTERN
          -- following the START index within the SOURCE text.  Returns
          -- zero if no match is found.  INDEX_ERROR can occur.
    ------------------------------------------------------------------------
    function CREATE (SOURCE : STRING) return TEXT;

          -- Converts the SOURCE string into TEXT.
    ------------------------------------------------------------------------
    procedure GET (ITEM : out TEXT);

          -- Reads a string from the user and converts it to TEXT.
    ------------------------------------------------------------------------
    procedure PUT (ITEM : in TEXT);

          -- Prints an object of TEXT.
    ------------------------------------------------------------------------
    procedure PUT_LINE (ITEM : in TEXT);

          -- Prints an object of TEXT and issues a new line.
    ------------------------------------------------------------------------
private

    type TEXT is ...

end BOUNDED_LENGTH_STRING;
```

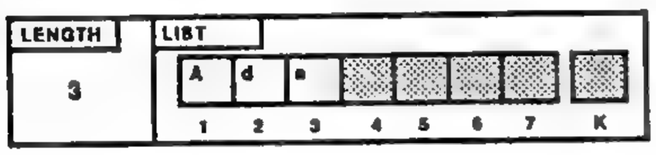
Privite type have no litah

Read in a string of TEXT and print it out one word per line. Assume that the string has no leading or trailing blanks and that there is precisely one blank between each word. Guard against the input of an empty string.

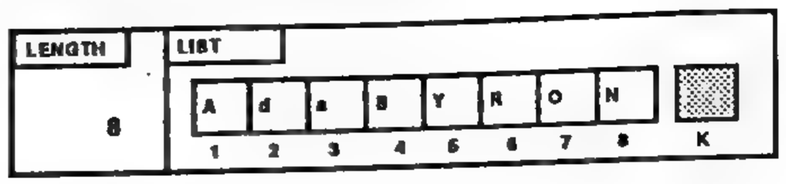
```
private

  LIST_TYPE is array (INDEX range <>) of CHARACTER;

  type TEXT is
     record
        LENGTH   : SIZE;
        LIST     : LIST_TYPE ( 1 .. MAX_SIZE);
     end record;

end BOUNDED_LENGTH_STRING;
```

-- K = ARBITRARY_MAXIMUM = 1000

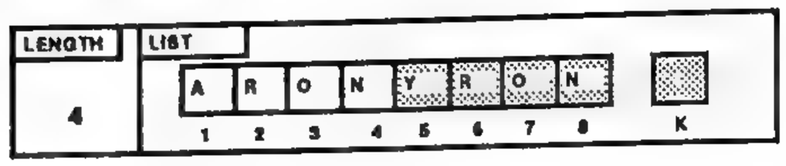
LADY : TEXT := CREATE ("Ada");



LADY := LADY & CREATE ("BYRON");



DELETE (LADY, 2, 4);



SPOT : INDEX := POS ("RO", LADY);

LNG : SIZE := LENGTH (LADY);

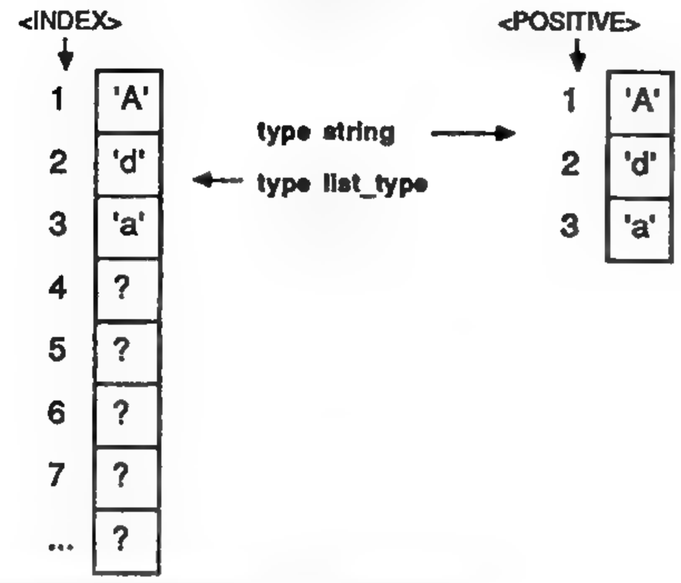

```
package body BOUNDED_LENGTH_STRING is

  -- The first two bodies are  proper bodies and are actually
  -- implemented by coverting the STRING to a TEXT via the CREATE
  -- routine and then calling the overloaded routines.  This satisfies the
  -- rule that all subunit names having the same ancestor library unit must
  -- be unique.

  procedure INSERT( SUB_TEXT  : in STRING;
                    ORIGINAL  : in out TEXT;
                    START     : in INDEX) is
  begin
     INSERT (CREATE  (SUB_TEXT), ORIGINAL, START);
  end INSERT;

  function POS(PATTERN : STRING; SOURCE : TEXT; START:INDEX := 1)
                  return INDEX is
   begin
      return POS (CREATE (PATTERN), SOURCE, START);
   end POS;

  -- The following function is included as a proper body because of the rule
  -- that the names of all compilation units must be identifiers.  If the body
  -- had been implemented as a body stub, then the corresponding subunit,
  -- a compilation unit, would be an operator symbol and not an identifier

  function "&"(HEAD : TEXT; TAIL : TEXT)  return TEXT is
      NEW_TEXT : TEXT;
  begin
      NEW_TEXT.LENGTH := HEAD.LENGTH + TAIL.LENGTH;
      NEW_TEXT.LIST (1 .. INDEX (NEW_TEXT.LENGTH)) :=
            HEAD.LIST (1 .. INDEX (HEAD.LENGTH))  &
            TAIL.LIST (1.. INDEX(TAIL.LENGTH));
      return NEW_TEXT;
   exception
      when constraint_error => raise SIZE_ERROR;
   end "&";

-- All other subprogram bodies can  be implemented  as body stubs
-- and could be inserted here.

end BOUNDED_LENGTH_STRING;
```



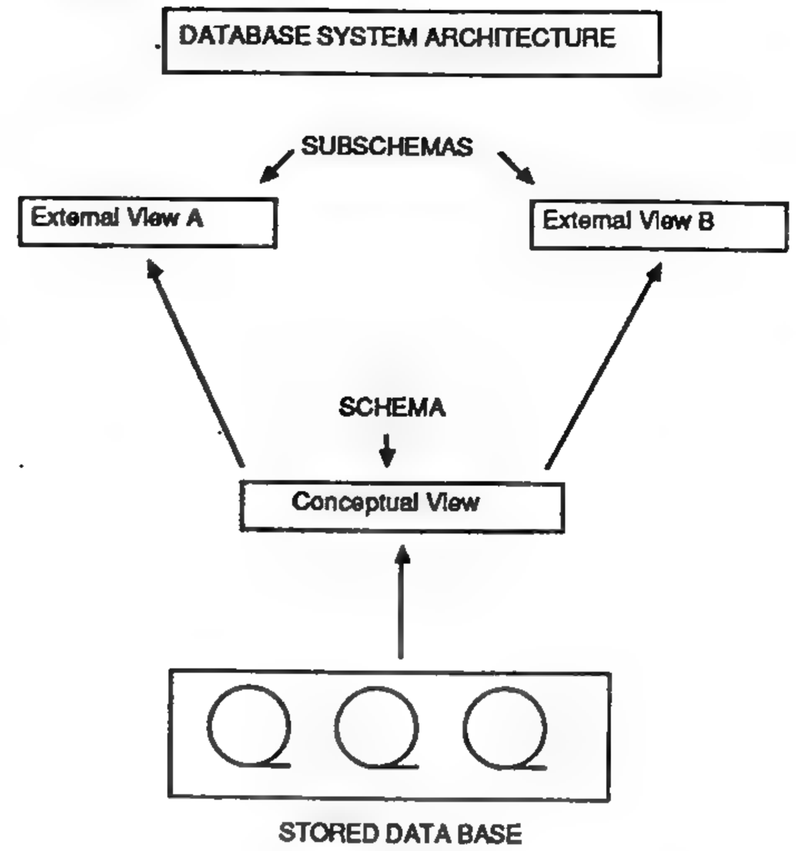
```
separate (BOUNDED_LENGTH_STRING)
procedure DELETE ( ORIGINAL : in out TEXT;
                   START    : INDEX;
                   COUNT    : SIZE) is


   TAIL_START : INDEX;
   TAIL_SIZE  : INDEX;

begin

     If START not in 1 .. INDEX (ORIGINAL.LENGTH) then
        raise INDEX_ERROR;
     end If;

     If COUNT > ORIGINAL.LENGTH - SIZE (START)  +1 then
        raise SIZE_ERROR;
     end If;

     TAIL_START  := START + INDEX(COUNT);
     TAIL_SIZE   := INDEX(ORIGINAL.LENGTH) - TAIL_START + 1;

     ORIGINAL.LIST (START .. START + TAIL_SIZE - 1)  :=
        ORIGINAL.LIST (TAIL_START .. TAIL_START + TAIL_SIZE - 1);

     ORIGINAL.LENGTH := ORIGINAL.LENGTH - COUNT;

end DELETE;
```

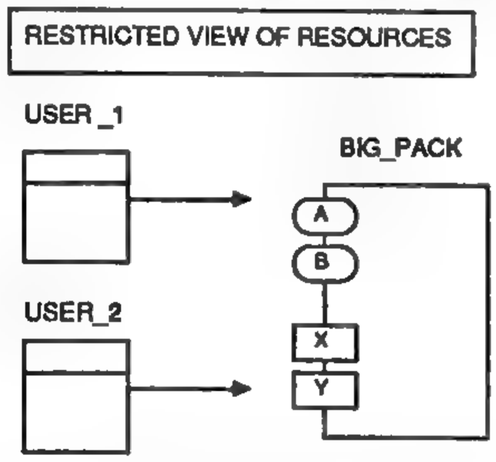
```
with TEXT_IO;
separate (BOUNDED_LENGTH_STRING)
procedure PUT (ITEM : In TEXT) is

begin

    TEXT_IO.PUT (STRING (ITEM.LIST (1 .. INDEX (ITEM.LENGTH))));

end PUT;
```

type conversion



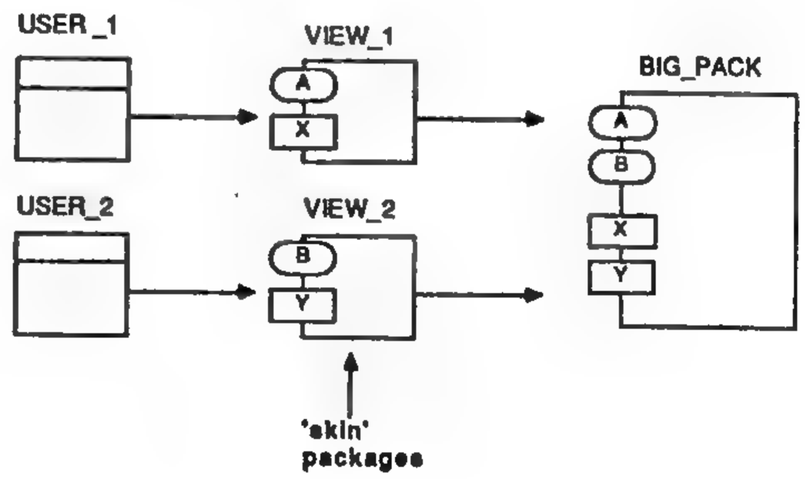

```
type STRING is array (POSITIVE range <>) of CHARACTER;

type LIST_TYPE is array (INDEX range <>) of CHARACTER;
```



- Q: IS IT POSSIBLE TO EXPORT ONLY TYPE A AND SUBPROGRAM X TO USER_1 AND TO EXPORT ONLY TYPE B AND SUBPROGRAM Y TO USER_2?


```
with BIG_PACK;
package VIEW_1 is
  type A is new BIG_PACK.A;
  procedure X ( . . .) renames BIG_PACK.X;
end VIEW_1;

with BIG_PACK;
package VIEW_2 is
  type B is new BIG_PACK.B;
  procedure Y ( . . .) renames BIG_PACK.Y;
end VIEW_2;
```

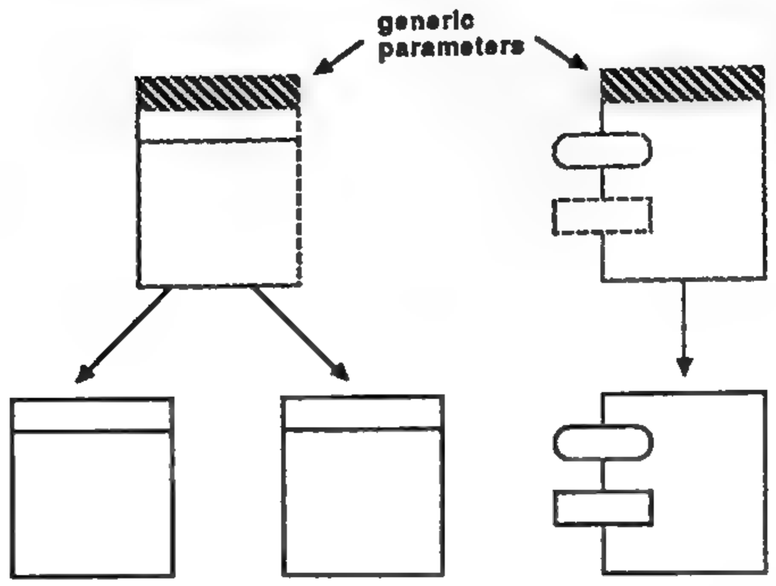


GENERIC PROGRAM UNITS

- DEFINE HIGH LEVEL TEMPLATES (MACROS)
- ALLOW Ada SUBPROGRAMS AND PACKAGES TO BE PARAMETERIZED
- ENCOURAGE DEVELOPMENT OF GENERAL PURPOSE LIBRARIES OF REUSEABLE SOFTWARE
- ALLOW TRANSLATION/ELABORATION TIME FACTORIZATION SIMILAR TO THE EXECUTION TIME FACTORIZATION ACHIEVED WITH SUBPROGRAMS

GENERIC PROGRAM UNITS

- A 'GENERIC DEFINITION' INCLUDES GENERIC PARAMETERS AND FORMS A PREFIX TO PROGRAM UNIT SPECIFICATIONS
- A 'GENERIC INSTANTIATION' CREATES A PROGRAM UNIT FROM A TEMPLATE
- GENERIC PARAMETERS CAN BE TYPES, VALUES, AND SUBPROGRAMS



Which is the smallest in each of the arrays?

```
type MY_LIST is array (1..5) of INTEGER;

THE_LIST : MY_LIST := (17, -4, 7, 0, 22);
```

THE_LIST

1	17
2	-4
3	7
4	0
5	22

```
SUBTYPE SHORT_WEEK IS DAYS range MON .. THU;
type WORK_TYPE is array (SHORT_WEEK) of CHARACTER;

THE_WEEK : WORK_TYPE := ('Q', 'A', 'D', 'S');
```

THE_WEEK

MON	'Q'
TUE	'A'
WED	'D'
THU	'S'


```
type ABC is ('A', 'B', 'C');
type DATE_LIST is array (ABC) of DATE_TYPE;

THE_DATES : DATE_LIST :=  ('A' => (4, JUL, 1776),
                           'B' => (19, JUN, 1963),
                           'C' => (1, DEC, 1822))
```

Q: Which is the 'smallest' date?

THE_DATES

'A'	DAY	4
	MONTH	JUL
	YEAR	1776
'B'	DAY	19
	MONTH	JUN
	YEAR	1963
'C'	DAY	1
	MONTH	DEC
	YEAR	1822

An algorithm for finding the smallest element in an integer array

```
procedure SAMPLE is

    type INDEX_SIZE is range 1 .. 5;

    type LIST is array (INDEX_SIZE) of INTEGER;

    function SMALLEST_INT (L : LIST) return INTEGER is

        RESULT : INTEGER := L (L'FIRST);

    begin

        for INDEX in L'RANGE
        loop

          if L(INDEX) < RESULT then

            RESULT := L(INDEX);

          end if;

        end loop;

        return RESULT;     -- send it back to caller

    end SMALLEST_INT;

begin -- SAMPLE

  ...

end SAMPLE;
```


GENERIC SPECIFICATION

```
generic

    type INDEX_TYPE is (<>);
    type BASE_TYPE is private;
    type ARRAY_TYPE is array (INDEX_TYPE) of BASE_TYPE;
    with function "<" (L, R : BASE_TYPE) return BOOLEAN is <>;

function LEAST (L : ARRAY_TYPE) return BASE_TYPE;
```

GENERIC BODY

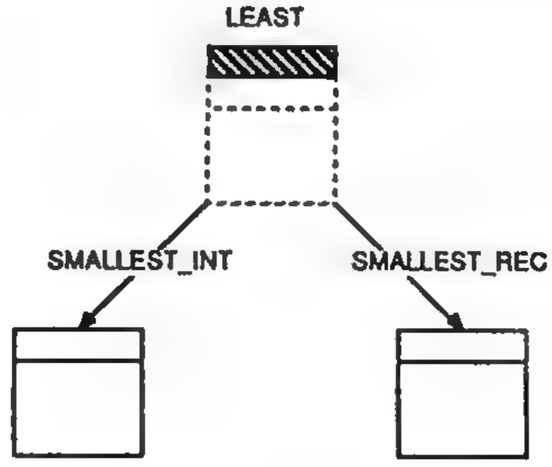
```
function LEAST (L : ARRAY_TYPE) return BASE_TYPE is

    RESULT : BASE_TYPE := L (L'FIRST);

begin

    for INDEX in L'range
    loop

      if L(INDEX) < RESULT then

        RESULT := L(INDEX);

      end if;

    end loop;

    return RESULT;

end LEAST;
```


A similar algorithm:

```
subtype ALPHA is character range 'a' . . 'f';

type MY_REC is
   record
      AGE          : NATURAL;
      GPA          : FLOAT;
      IS_RESIDENT  : BOOLEAN;
   end record;

type STUDENTS is array (ALPHA) of MY_REC;

-- But, "<" is not a primitive operation on record types.
-- Therefore, we must provide the capability.  In
-- this case we will define a 'less-than' operation on
-- the age components of the records.

function LESS (X, Y : MY_REC) return BOOLEAN is
begin
      return X.AGE < Y.AGE;
end;
```

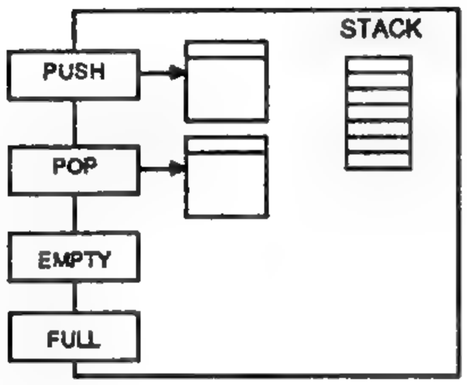

GENERIC INSTANTIATIONS

```
function SMALLEST_INT is new LEAST (INDEX_TYPE   => INDEX_SIZE,
                                    BASE_TYPE    => INTEGER,
                                    ARRAY_TYPE   => LIST);


function SMALLEST_REC is new LEAST (INDEX_TYPE   => ALPHA,
                                    BASE_TYPE    => MY_REC,
                                    ARRAY_TYPE   => STUDENTS,
                                      "<"        => LESS);
```



A GENERIC STACK PACKAGE

```
generic

   type ELEMENT is private;

package STACK_PACK is

    procedure PUSH (OBJECT : In   ELEMENT);
    procedure POP   (OBJECT : out ELEMENT);

    EMPTY, FULL : exception;

end STACK_PACK;
```




```
package body STACK_PACK is

   MAX   : constant := 100;
   TOP   : NATURAL := 0;
   STACK : array (1 .. MAX) of ELEMENT;

   procedure PUSH (OBJECT : In ELEMENT) is
   begin

      If TOP = MAX then
         raise FULL;
      end if;

      TOP := TOP + 1;
      STACK (TOP) := OBJECT;

   end PUSH;


   procedure POP (OBJECT : out ELEMENT) is
   begin

      If TOP = 0 then
         raise EMPTY;
      end if;

      OBJECT := STACK (TOP);
      TOP := TOP - 1;

   end POP;

end STACK_PACK;
```


GENERIC TYPE PARAMETERS

- TO MATCH ANY TYPE (NO OPERATIONS)
  type <ident> is limited private;
- TO MATCH ANY TYPE PERMITTING ASSIGNMENT AND TEST FOR (IN)EQUALITY
  type <ident> is private;
- TO MATCH AN ACCESS TYPE
  type <ident_1> is access <ident_2>;
- TO MATCH ANY DISCRETE TYPE
  type <ident> is (<>);
- TO MATCH NUMERIC TYPES
  type <ident> is range <>;
  type <ident> is delta <>;
  type <ident> is digits <>;
- TO MATCH ANY CONSTRAINED ARRAY
  type <ident_1> is array(<ident_2>) of <ident_3>;
- TO MATCH ANY UNCONSTRAINED ARRAY
  type <id_1> is array (<id_2> range <>) of <id_3>;

GENERIC OBJECT PARAMETERS

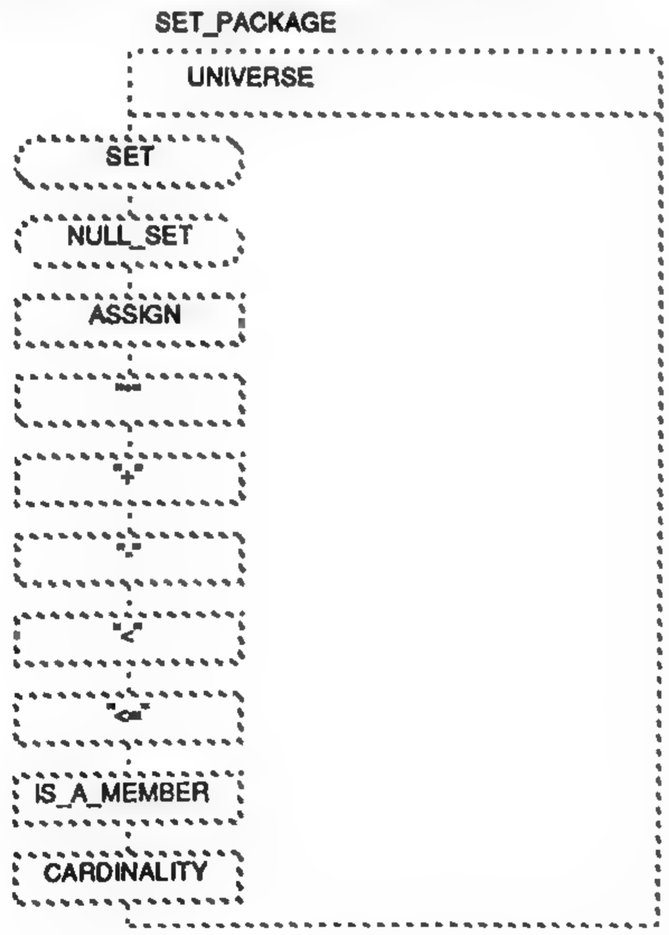
```
generic
   MAX : In POSITIVE;     -- generic formal OBJECT parameter
   type ELEMENT is private;
package STACK_PACK is
  procedure PUSH (OBJECT : in    ELEMENT);
  procedure POP   (OBJECT : out  ELEMENT);
  EMPTY, FULL : exception;
end STACK_PACK;

package body STACK_PACK is

  TOP    : NATURAL := 0;
  STACK : array (1 .. MAX) of ELEMENT;

  procedure PUSH . . .
  procedure POP . . .

end STACK_PACK;

      -- generic instantiations

package INT_STACK is new STACK_PACK
                          (MAX => 50, ELEMENT => INTEGER);

package CHAR_STACK is new STACK_PACK
                                   (100, CHARACTER);
```

DEVELOP A GENERIC SET CAPABILITY

- SETS ARE DRAWN FROM SOME DISCRETE UNIVERSE
- SETS CAN BE ASSIGNED VALUES
- THE UNION OF TWO SETS IS A THIRD SET CONTAINING ALL ELEMENTS WHICH ARE IN EITHER THE FIRST SET OR THE SECOND SET
- THE INTERSECTION OF TWO SETS IS A THIRD SET WHICH CONTAINS ALL ELEMENTS WHICH ARE IN BOTH THE FIRST SET AND THE SECOND SET
- THE DIFFERENCE BETWEEN TWO SETS IS A THIRD SET WHICH CONTAINS ALL ELEMENTS WHICH ARE IN THE FIRST SET AND NOT IN THE SECOND SET
- A SET 'A' IS A COMMON SUBSET OF A SET 'B' IF AND ONLY IF 'A' IS EQUAL TO THE INTERSECTION OF 'A' AND 'B'
- A SET 'A' IS A PROPER SUBSET OF A SET 'B' IF AND ONLY IF 'A' IS A COMMON SUBSET OF 'B' AND 'A' IS NOT EQUAL TO 'B'
- FOR EVERY ELEMENT 'e' OF A GIVEN UNIVERSE AND SET 'S' OF THE SAME UNIVERSE, EITHER 'e' IS A MEMBER OF S OR 'e' IS NOT A MEMBER OF 'S'
- THE CARDINALITY OF A SET IS THE NUMBER OF ELEMENTS CURRENTLY IN THE SET
- THE NULL SET IS THE SET CONTAINING NO ELEMENTS

OBJECTS AND OPERATIONS

- SET
  - -- assignment
  - -- (In)equality =
  - -- Intersection *
  - -- Union +
  - -- Difference -
  - -- Proper Subset <
  - -- Common Subset <=
  - -- Membership
  - -- Cardinality
- NULL_SET
- UNIVERSE




```
generic
   type UNIVERSE is (<>);
package SET_PACKAGE is

   type SET is private;
   NULL_SET : constant SET;      -- deferred

   function ASSIGN (ELEMENT : UNIVERSE) return SET;
   function ASSIGN (FROM,TO  : UNIVERSE) return SET;

   function "=" (SET_1, SET_2 : SET) return SET;

   function "/=" (SET_1, SET_2 : SET) return SET;

   function "+" (SET_1     : SET;
                 ELEMENT : UNIVERSE) return SET;

   function "+" (ELEMENT : UNIVERSE;
                 SET_1     : SET) return SET;

   function "-" (SET_1, SET_2 : SET) return SET;

   function "-" (SET_1     : SET;
                 ELEMENT : UNIVERSE) return SET;
   function "<"  (SET_1, SET_2 : SET) return BOOLEAN;

   function "<=" (SET_1, SET_2 : SET) return BOOLEAN;

   function IS_A_MEMBER (ELEMENT : UNIVERSE;
                         OF_SET : SET) return BOOLEAN;

   function CARDINALITY (S :SET) return NATURAL;

private
   ...
end SET_PACKAGE;
```

GENERIC PACKAGE BODY

- Indistinguishable from a routine package body except that all reference is to generic parameters
- Can take full advantage of actual private type implementation. The type is not really 'private' to the implementor.

```
package body SET_PACKAGE is

   -- all bodies of subprograms whose specification
   -- appeared in the package spec must be included
   -- here. They could be included as stubs and then
   -- be completed as subunits and separately compiled.

end SET_PACKAGE;
```


LOGICAL OPERATIONS ON BOOLEAN ARRAYS

- The logical operations NOT, AND, OR and XOR are just as appropriate for one-dimensional arrays whose component type is 'boolean' as they are for scalar objects of type 'boolean'.

```
type BOOLS is array (1 .. 4) of BOOLEAN;

T : constant BOOLEAN := TRUE;
F : constant BOOLEAN := FALSE;

A : BOOLS := (T, T, F, F);
B : BOOLS := (T, F, T, F);
```

	A	B	not A	A and B	A or B	A xor B
1	T	T	F	T	T	F
2	T	F	F	F	T	T
3	F	T	T	F	T	T
4	F	F	T	F	F	F

- THIS CAPABILITY (BOOLEAN OPERATIONS ON BOOLEAN ARRAYS) LEADS US TO A VERY NATURAL DATA STRUCTURE FOR SETS

```
private

   type SET is array (UNIVERSE) of BOOLEAN;
   NULL_SET : constant SET := (others => FALSE);

end SET_PACKAGE;
```

- Consider the following application:

```
type NORDEN is (DK, S, N, SF);
package NORTH_SET is new SET_PACKAGE (NORDEN);
use NORTH_SET;
A, B, C, SCANDINAVIA : NORTH_SET.SET;
...
A := ASSIGN (FROM => DK, TO => S);
B := ASSIGN (DK) + N;
C := A * B;
SCANDINAVIA := A + B;
```

	A	B	C	SCANDINAVIA
DK	T	T	T	T
S	T	F	F	T
N	F	T	F	T
SF	F	F	F	F
	{DK,S}	{DK,N}	{DK}	{DK,S,N}

IMPLEMENTATION

```
function "*" (SET_1, SET_2 : SET) return SET is
begin
      return (SET_1 and SET_2);
end;


function "+" (SET_1, SET_2 : SET) return SET is
begin
      return (SET_1 or SET_2);
end;


function "+" (SET_1: SET;
              ELEMENT : UNIVERSE) return SET is

  RESULT : SET := SET_1;

begin
      RESULT (ELEMENT) := TRUE;
      return RESULT;
end;


function "+" (ELEMENT : UNIVERSE;
              SET_1 : SET) return SET is
begin
    return SET_1 + ELEMENT;
end;
```



```
function "-" (SET_1, SET_2 : SET) return SET is
begin
      return (SET_1 and (not SET_2));
end;


function "-" (SET_1: SET;
              ELEMENT : UNIVERSE) return SET is

  RESULT : SET := SET_1;

begin
      RESULT (ELEMENT) := FALSE;
      return RESULT;
end;


function "<=" (SET_1, SET_2 : SET) return BOOLEAN is
begin
      return SET_1 = SET_1 * SET_2;
end;


function "<" (SET_1, SET_2 : SET) return BOOLEAN is
begin
      return (SET_1 <= SET_2) and (SET_1 /= SET_2);
end;
```



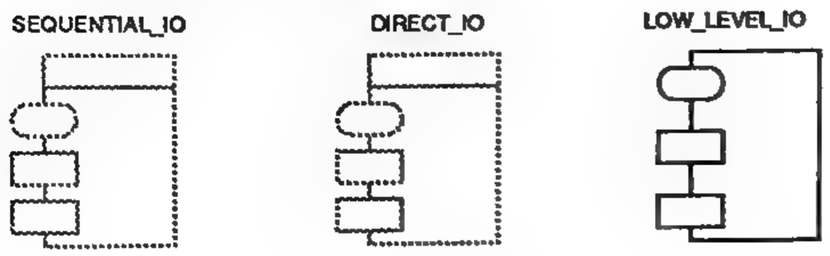
```
function IS_A_MEMBER (ELEMENT : UNIVERSE;
                      OF_SET : SET) return BOOLEAN is
begin
      return OF_SET (ELEMENT);
end;


function CARDINALITY (S : SET) return NATURAL is

  TOTAL : NATURAL := 0;

begin

      for INDEX in UNIVERSE
      loop

         if S (INDEX) then
            TOTAL := TOTAL + 1;
         end if;

      end loop;

      return TOTAL;

end CARDINALITY;
```


- The assignment (replacement) operation (:=) is allowed since type SET is private and not limited private
- The ability to assign an element or a range of elements to a set is also helpful

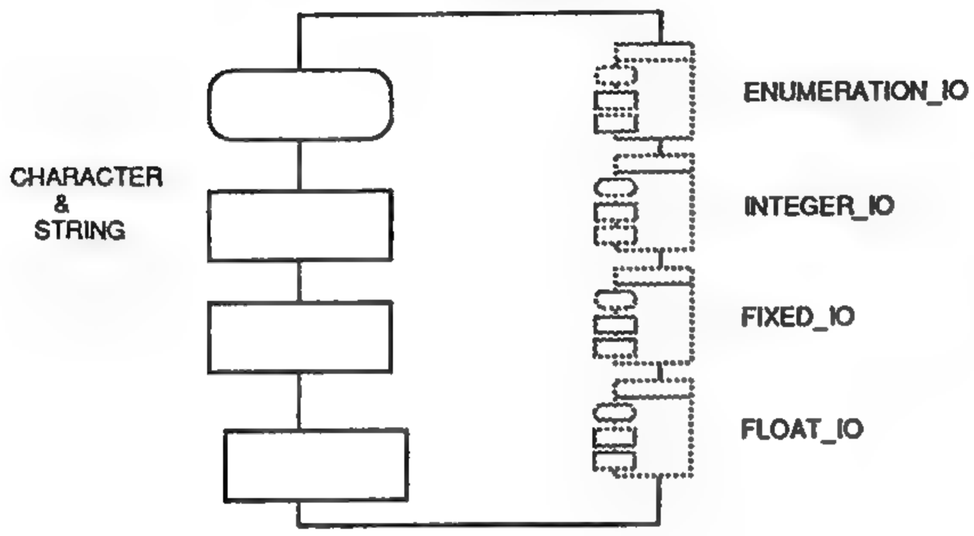
```
function ASSIGN (ELEMENT : UNIVERSE) return SET is
begin
   return (ELEMENT => TRUE, others => FALSE);
end


function ASSIGN (FROM, TO : UNIVERSE) return SET is
begin

    return (FROM .. TO => TRUE, others => FALSE);

end ASSIGN;
```

INPUT/OUTPUT

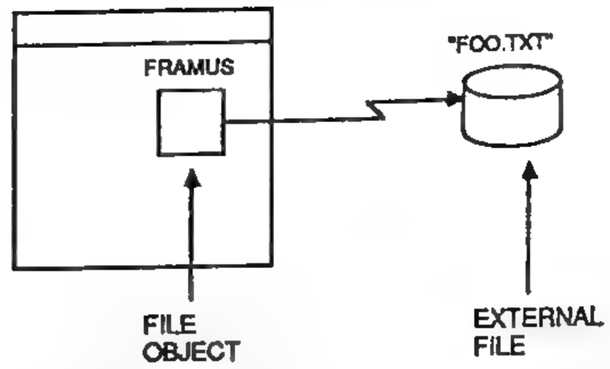
- IN ADA, I/O IS HANDLED VIA PACKAGES WHICH COME WITH THE LANGUAGE



TEXT_IO



FILE OBJECTS




```
type FILE_TYPE is limited private;
type FILE_MODE is (IN_FILE, OUT_FILE);

procedure CREATE ( FILE   : in out FILE_TYPE;
                   MODE   : in FILE_MODE := default;
                   NAME   : in STRING := "";
                   FORM   : in STRING := "");

procedure  OPEN ( FILE    : in out FILE_TYPE;
                  MODE    : in FILE_MODE;
                  NAME    : in STRING;
                  FORM    : in STRING := "");
```

-- opening a file:

```
...
FRAMUS : TEXT_IO.FILE_TYPE;                          -- Declaration
...
TEXT_IO.OPEN (FRAMUS, TEXT_IO.IN_FILE, "FOO.TXT");   -- Statement
```


OPERATIONS ON FILE OBJECTS

- OPERATIONS ON ALL FILES

procedures	functions	
CREATE	MODE	-- FILE_MODE
OPEN	NAME	-- STRING
CLOSE	FORM	-- STRING
DELETE	IS_OPEN	-- BOOLEAN
RESET	END_OF_FILE	-- BOOLEAN

- OPERATIONS ON SEQUENTIAL AND DIRECT FILES ONLY

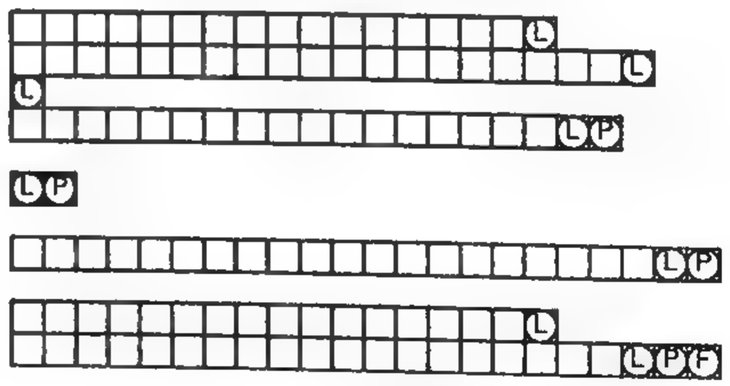
procedures
READ
WRITE

- OPERATIONS ON DIRECT FILES ONLY

procedures	functions	
SET_INDEX	INDEX	-- POSITIVE_COUNT
	SIZE	-- COUNT (FROM 0)

package TEXT_IO

- PROVIDES I/O FOR CHARACTERS AND STRINGS
- CONTAINS GENERIC PACKAGES FOR: ENUMERATION_IO, FIXED_IO, FLOAT_IO, INTEGER_IO
- FILE LAYOUT
  - -- A file is a sequence of pages (numbered from 1)
  - -- A page is a sequence of lines (numbered from 1)
  - -- A line is a sequence of characters (columns)



SOURCE: 'Ada as a second language' by Norman H. Cohen McGraw-Hill, 1986.

STANDARD FILES

- IMPLEMENTATION DEFINED
- INPUT (USUALLY KEYBOARD)
- OUTPUT (USUALLY CRT)

DEFAULT FILES

- INITIALLY, THE STANDARD FILES
- CAN BE CHANGED DURING EXECUTION
- I/O OPERATIONS CAN NAME A SPECIFIC FILE OR CAN RELY ON THE DEFAULT FILE

TEXT_IO OPERATIONS

- OPERATIONS ON OUT_FILE

procedures	functions
PUT	LINE_LENGTH
SET_LINE_LENGTH	PAGE_LENGTH
NEW_LINE	COL
NEW_PAGE	LINE
SET_COL	PAGE
SET_LINE	
SET_PAGE	

- OPERATIONS ON IN_FILE

procedures	functions
SKIP_LINE	END_OF_LINE
SKIP_PAGE	END_OF_PAGE
SET_COL	COL
SET_LINE	LINE
GET	PAGE

- OPERATIONS FOR I/O OF STRINGS ONLY

procedures
GET_LINE
PUT_LINE

I/O FOR OTHER SCALAR TYPES

- ASSUME THE FOLLOWING TYPE DECLARATIONS

```
type GENDER is (MALE, FEMALE);
type SIZE is range 1 .. 10;
```

- THESE INSTANTIATIONS ARE NECESSARY IN ORDER TO HAVE I/O

```
package GENDER_IO is new
                TEXT_IO.ENUMERATION_IO(GENDER);
package SIZE_IO is new
                TEXT_IO.INTEGER_IO(SIZE);
```


I/O EXCEPTIONS

```
package IO_EXCEPTIONS is
   STATUS_ERROR    : exception;
   MODE_ERROR      : exception;
   NAME_ERROR      : exception;
   USE_ERROR       : exception;
   DEVICE_ERROR    : exception;
   END_ERROR       : exception;
   DATA_ERROR      : exception;
   LAYOUT_ERROR    : exception;
end IO_EXCEPTIONS;
```

I/O PACKAGES USE renames TO EXPORT EXCEPTIONS

```
with IO_EXCEPTIONS;
package TEXT_IO is
...
 USE_ERROR : exception renames
     IO_EXCEPTIONS.USE_ERROR;
...
end TEXT_IO;
```

SAMPLE I/O PROGRAM

- THE FOLLOWING PROGRAM READS INTEGERS FROM AN EXISTING FILE ("FOO.TXT"), CALCULATES THE SUM AND OUTPUTS THE SINGLE INTEGER RESULT TO A NEW FILE ("RESULT.TXT")

```
with TEXT_IO;
procedure SUM_UP is

  package INT_IO is new TEXT_IO.INTEGER_IO(INTEGER);

  INPUT_NUMBERS : TEXT_IO.FILE_TYPE;
  RESULT        : TEXT_IO.FILE_TYPE;
  SUM           : INTEGER := 0;
  NUMBER        : INTEGER;

begin
  TEXT_IO.OPEN (INPUT_NUMBERS, TEXT_IO.IN_FILE, "FOO.TXT");
  TEXT_IO.CREATE (RESULT, TEXT_IO.OUT_FILE, "RESULT.TXT");

  while not TEXT_IO.END_OF_FILE (INPUT_NUMBERS)
    loop
      INT_IO.GET (INPUT_NUMBERS, NUMBER);
      SUM := SUM + NUMBER;
    end loop;

  INT_IO.PUT (RESULT, SUM);

  TEXT_IO.CLOSE (INPUT_NUMBERS);
  TEXT_IO.CLOSE (RESULT);

end SUM_UP;
```


TEXT_IO FORMAT OPTIONS

```
INT_IO.PUT (17, WIDTH => 5);                  -- bbb17
INT_IO.PUT (17, BASE => 8);                   -- 8#21#
FLT_IO.PUT (17.5, FORE => 3, AFT => 2);       -- b17.50
FLT_IO.PUT (17.5, EXP => 3);                  -- 1.75E+01
ENUM_IO.PUT (NORMAL, WIDTH => 8);             -- NORMALbb
ENUM_IO.PUT (DOWN,  LOWER_CASE);              -- down
TEXT_IO.PUT ('A');                            -- A
package CHAR_IO is new
            TEXT_IO.ENUMERATION_IO (CHARACTER);
CHAR_IO.PUT ('A');                            -- 'A'
```


the GET_LINE operation

- A routine to input two words and output them one word per line (assumes exactly two words)

```
with TEXT_IO;
procedure IO_SAMPLE is

  SOURCE : STRING (1 .. 60);
  SPOT   : NATURAL;
  COUNT  : NATURAL;

begin

  TEXT_IO.GET_LINE (SOURCE, COUNT);

  SPOT := 1;
  loop
     exit when SOURCE (SPOT) = ' ';
     SPOT := SPOT + 1;
  end loop;

  TEXT_IO.PUT_LINE (SOURCE (1 .. SPOT - 1));
  TEXT_IO.PUT_LINE (SOURCE (SPOT+1 .. COUNT));

end IO_SAMPLE;
```

TASKS

ENTITIES WHOSE EXECUTIONS PROCEED IN PARALLEL

CAN BE CONSIDERED TO EXECUTE ON THEIR OWN LOGICAL PROCESSOR

DIFFERENT TASKS PROCEED INDEPENDENTLY, EXCEPT AT POINTS WHERE THEY SYNCHRONIZE

VARIOUS ACTUAL IMPLEMENTATIONS

- -- MULTICOMPUTERS
- -- MULTIPROCESSORS
- -- INTERLEAVED EXECUTION

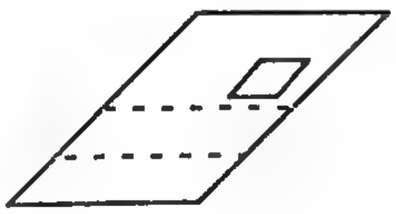
TASK CONSIDERATIONS

- HOW IS A TASK ACTIVATED?
- HOW IS A TASK TERMINATED?
- HOW DO TASKS COMMUNICATE?
- WHAT ABOUT DEADLOCK?
- CAN A TASK 'TIME OUT'?
- IS THERE A PRIORITY SCHEME?
- HOW IS 'SHARED' DATA PROTECTED?
- DO Ada TASKS ISSUE OPERATING SYSTEM CALLS?
- HOW DO EXCEPTIONS AFFECT TASKS?

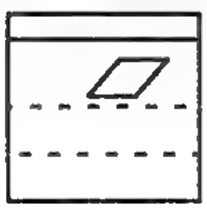
TASK DEPENDENCE

- EACH TASK DEPENDS ON AT LEAST ONE MASTER
- A MASTER CAN BE
  - -- A TASK
  - -- A BLOCK STATEMENT
  - -- A SUBPROGRAM
  - -- A LIBRARY PACKAGE

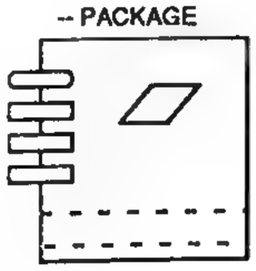
--TASK



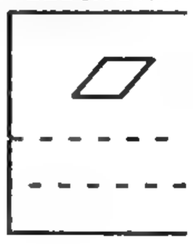
-- SUBPROGRAM



-- PACKAGE



-- BLOCK STMT



TASK ACTIVATION

A TASK DECLARED IN A <declarative_part> OF A SUBPROGRAM, TASK, PACKAGE OR BLOCK STATEMENT IS ACTIVATED

AFTER THE PARENT IS ELABORATED AND BEFORE THE PARENT BEGINS EXECUTION

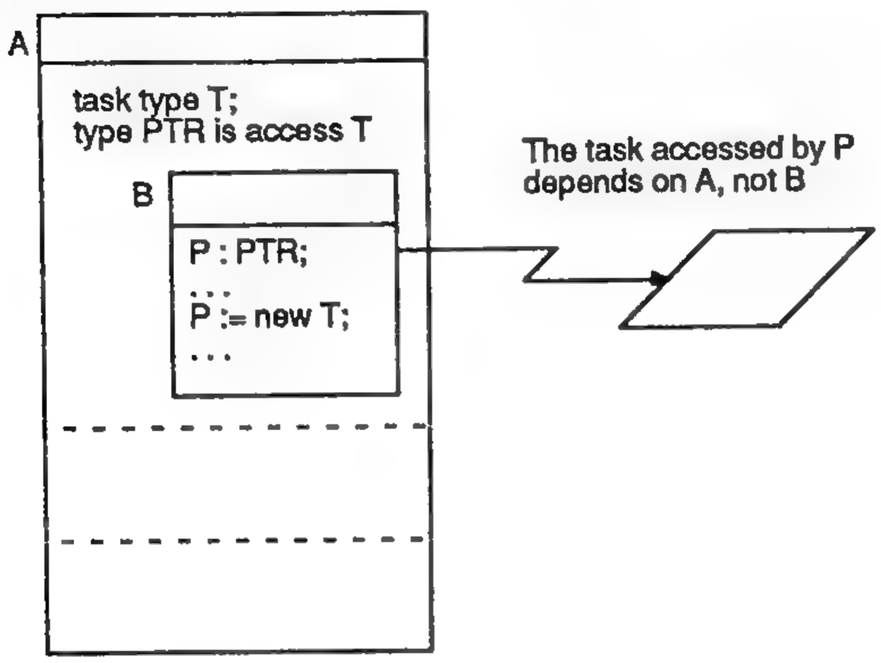
A TASK WHOSE SPECIFICATION APPEARS IN A PACKAGE SPECIFICATION IS ACTIVATED

AFTER THE PACKAGE BODY IS ELABORATED

DYNAMIC TASK ACTIVATION

A TASK CAN BE ACTIVATED DYNAMICALLY VIA AN ALLOCATOR.

THE MASTER OF THE ALLOCATED TASK IS THE UNIT WHICH CONTAINS THE ACCESS TYPE DECLARATION (NOT THE UNIT THAT EXECUTED THE ALLOCATOR).



TASK TERMINATION

- COMPLETION OF EXECUTION
  - -- A task, block statement or subprogram is completed when its sequence of statements has been executed.
  - -- A block statement is completed when it reaches a goto, exit, or return transferring control out of the block statement.
  - -- A procedure or function is completed upon executing a return.
  - -- A task, block statement or subprogram is completed when an exception is raised and there is no handler or, after handling the exception.
- TERMINATION OF TASKS
  - -- A task with no dependent tasks terminates upon completion.
  - -- A task with dependents terminates when it is completed and all its children are terminated.
  - -- A block statement or subprogram which is complete is not left until all of its children tasks are terminated.


```
procedure MAIN is

    task type T;

    type T_PTR is access T;

    procedure P is separate;

    task body T is separate;

begin

    for INDEX in 1 .. 3 loop
        P;    -- A call to procedure P
    end loop;

end MAIN;
```



```
separate (MAIN)
procedure P is          -- version 1

    T_ARRAY : array (1 .. 3) of T;

begin

    ...

end P;
```

```
separate (MAIN)
procedure P is          -- version 2

    T_ARRAY: array (1 .. 3) of T_PTR;

begin

    for I in 1 .. 3 loop
        T_ARRAY (I) := new T;
    end loop;

end P;
```

HOW MANY TASKS ARE ACTIVE AT ONCE?

TASK ENTRIES

TASKS COMMUNICATE VIA CHANNELS CALLED ENTRIES.

AN ENTRY OF A TASK IS ANALOGOUS TO A SUBPROGRAM OF A PACKAGE.

<task_specification> ::=

```
task [type] <identifier>     [is
    {<entry_declaration>}
    {<representation_clause>}

end [<task_simple_name>]  ];
```

<entry_declaration> ::=

```
entry <identifier> [(<discrete_range>)] [<formal_part>];
```


SAMPLE TASK SPECIFICATIONS

```
task SERVER;
-----------------------------
task type SWITCH is
  entry PORT (LOW .. HIGH)(N : INTEGER);
end;
-----------------------------
task PROTECTED_STACK is
  pragma PRIORITY (17);
  entry POP  (OBJECT : out FLOAT);
  entry PUSH (OBJECT : in  FLOAT);
end PROTECTED_STACK;
-----------------------------
task BEAN is
  entry COUNTER (N : in INTEGER);
  for COUNTER use at 16#1FF#;
end BEAN;
```


CALLING AN ENTRY

- TASK SPECIFICATION

```
task PROTECTED_STACK is
  entry POP  (OBJECT : out FLOAT);
  entry PUSH (OBJECT : in FLOAT);
end PROTECTED_STACK;
```

- ENTRY CALLS -- must name the task

```
PROTECTED_STACK.PUSH (3.1415);
PROTECTED_STACK.POP (MY_FLOAT);
```

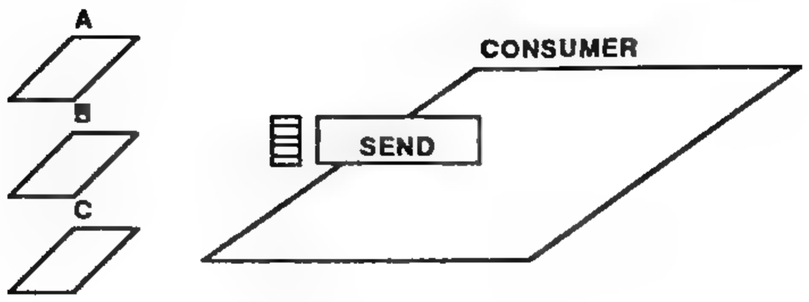
- AN ENTRY CAN BE RENAMED AS A PROCEDURE

```
procedure POP (OBJECT : out FLOAT) renames
            PROTECTED_STACK.POP;
```


ENTRY QUEUES

- THERE IS AN IMPLICIT QUEUE ASSOCIATED WITH EACH ENTRY.
- THE FIRST TASK TO CALL AN ENTRY WILL BE THE FIRST TASK TO RENDEZVOUS.
- ALL OTHER TASKS WAIT IN THE QUEUE IN ORDER OF ARRIVAL.
- IT IS POSSIBLE TO LEAVE A QUEUE BEFORE BEING SERVED.
- A TASK CAN BE IN ONLY ONE QUEUE AT A TIME.

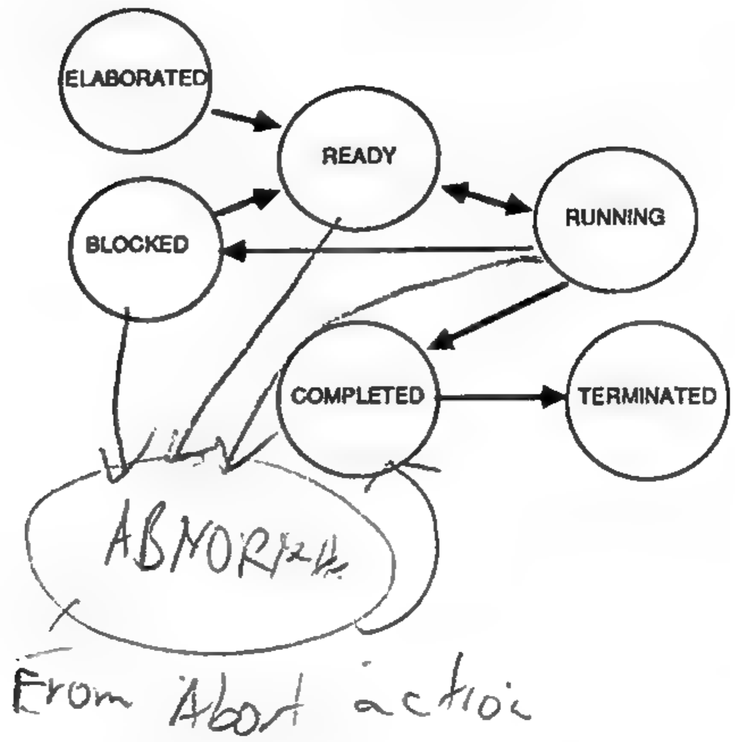
ENTRY QUEUES




```
procedure MAIN is
   task type PRODUCER;

   A, B, C : PRODUCER;

   task CONSUMER is
      entry SEND (N : in INTEGER);
   end CONSUMER;

   task body PRODUCER is separate;

   task body CONSUMER is separate;

begin
   ...
end MAIN;
```


TASK STATES

- ELABORATED -- declarations now exist
- RUNNING -- currently assigned a processor
- READY -- unblocked, waiting for a processor
- BLOCKED -- delayed or waiting for rendezvous
- COMPLETED -- task has reached its 'end'
- TERMINATED -- all of tasks children have terminated



TASK PRIORITY

- STATIC VALUE
- SET WITH A PRAGMA
- ALLOWS THE TASK WITH HIGHEST PRIORITY TO MOVE FROM 'READY' TO 'RUNNING' AND, IF NEED BE, TO PREEMPT A LOWER PRIORITY TASK
- DOES NOT AFFECT THE ORDER IN WHICH A QUEUED TASK WILL BE SERVED

```
task HIGH_PRIORITY is
   pragma PRIORITY (7);
   entry ...
end HIGH_PRIORITY;
```

" If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processors and the same other processing resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not."

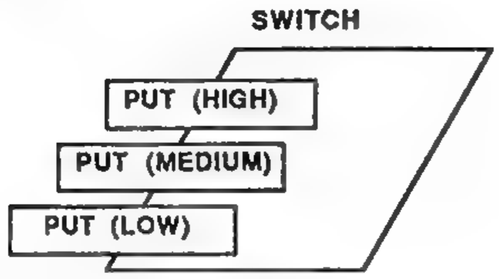
TASK ASYMMETRY

- A CALLING TASK MUST KNOW NAME OF CALLED TASK AND NAME OF ENTRY (LIKE NEEDING TO KNOW PHONE NUMBER WHEN YOU CALL).
- A CALLED TASK DOES NOT KNOW THE NAME OF THE CALLER (LIKE ANSWERING THE PHONE).
- SEPARATION OF SPECIFICATION FROM BODY ALLOWS MUTUAL CALLING OF TASKS.
- A TASK CAN CALL ITSELF (BUT, DEADLOCK OCCURS).

FAMILIES OF ENTRIES

- A SET OF PEER ENTRIES
- INDEXED BY A DISCRETE VALUE
- A 'ONE-DIMENSIONAL ARRAY' OF ENTRIES

```
type IMPORTANCE is (LOW, MEDIUM, HIGH);
task SWITCH is
   entry PUT (IMPORTANCE)(MSG : in string);
end SWITCH;
```



- CALLING A FAMILY MEMBER

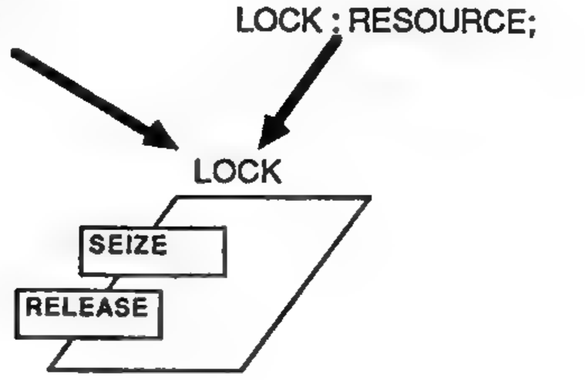
```
SWITCH.PUT (LOW) (NEW_MESSAGE);
```


TASK TYPES

- TASK TYPES ARE LIMITED PRIVATE
  -- no assignment
  -- no test for (in)equality

```
task LOCK is
  entry SEIZE;
  entry RELEASE;
end LOCK;
```

```
task type RESOURCE is
  entry SEIZE;
  entry RELEASE;
end RESOURCE;

LOCK : RESOURCE;
```



TASK OBJECT DECLARATIONS

```
type PROTECTED is
        record
            OBJECT : FLOAT;
            KEY    : RESOURCE;
        end record;

SAFE : PROTECTED;

LOCK : RESOURCE;

COLLECTION : array (1 .. 10) of RESOURCE;

type PTR is access RESOURCE;
GUARD : PTR;
...
GUARD := new RESOURCE;  -- an allocator
```

TASK ENTRY CALLS

```
SAFE.KEY.SEIZE;

LOCK.RELEASE;

COLLECTION (8).SEIZE;

GUARD.RELEASE;
```


ATTRIBUTES OF TASKS

- T'CALLABLE
  -- Yields the value false when the task T is completed or terminated or aborted

- T'TERMINATED
  -- Yields the value true if the task T is terminated

- E'COUNT
  -- Yields the number of entry calls presently queued on the entry E. Does not include the task which is currently in rendezvous

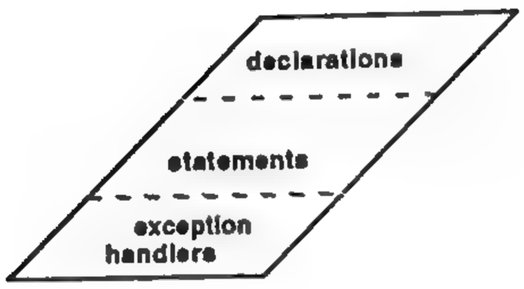
TASK BODIES

```
<task body> ::=
    task body <task_simple_name> is
        [ <declarative_part> ]
    begin
        <sequence_of_statements>
    [ exception
        <exception_handler>
        {<exception_handler>} ]
    end [ <task_simple_name> ];
```


TASK BODIES

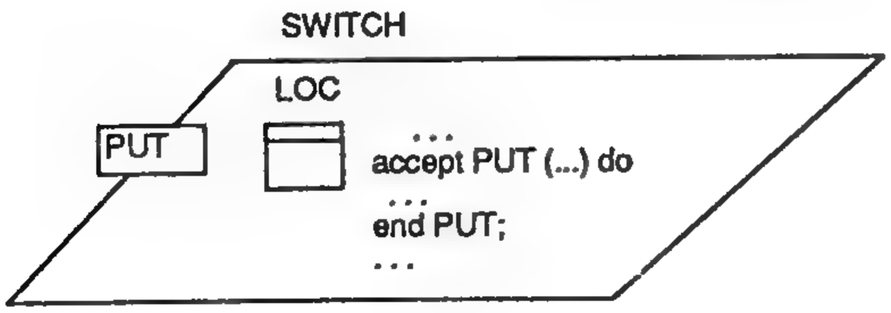
- MAY BE SEPARATELY COMPILED
- MAY CONTAIN ACCEPT AND SELECT STATEMENTS (AS WELL AS OTHERS)

```
task body RESOURCE is
   ...
begin
   ...
exception
   ...
end RESOURCE;
```



ACCEPT STATEMENTS

- ALWAYS CORRESPOND TO TASK ENTRIES
- CAN DEFINE A SEQUENCE OF STATEMENTS TO BE EXECUTED DURING RENDEZVOUS WITH A CALLING TASK
- MUST APPEAR DIRECTLY IN THE TASK BODY (NOT IN A NESTED SUBPROGRAM)
- MUST NOT APPEAR WITHIN ANOTHER ACCEPT STATEMENT FOR THE SAME ENTRY OR FAMILY OF ENTRIES



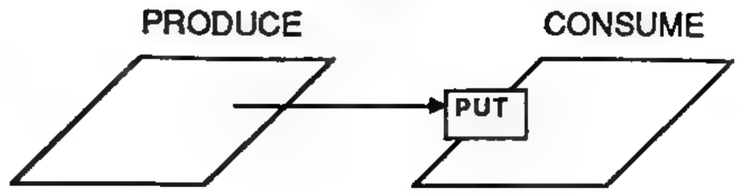
ACCEPT STATEMENTS

```
<accept_statement> ::=
    accept <entry_simple_name>
        [(<entry_index)] [formal_part] [do
        <sequence_of_statements>
    end [<entry_simple_name>] ];
```

RENDEZVOUS

- THE INTERACTION THAT OCCURS BETWEEN TWO PARALLEL TASKS WHEN ONE TASK HAS CALLED AN ENTRY OF THE OTHER TASK, AND A CORRESPONDING ACCEPT STATEMENT IS BEING EXECUTED BY THE CALLED TASK ON BEHALF OF THE CALLING TASK.
- FOR SIMPLE RENDEZVOUS, WHICHEVER TASK ARRIVES AT THE RENDEZVOUS POINT FIRST WILL GO INTO A SLEEPING WAIT.
- DURING RENDEZVOUS, THE TWO TASKS ARE LOCKED TOGETHER.
- UPON COMPLETION OF RENDEZVOUS, THE TWO TASKS CONTINUE IN PARALLEL.

I wait for a — caller & one
wait-time — caller & several
wait-% — — immediately / ended



- TASK SPECIFICATIONS

```
task PRODUCE;

task CONSUME is
   entry PUT (N : INTEGER);
end CONSUME;
```

- TASK RENDEZVOUS

```
task body PRODUCE is
...
CONSUME.PUT (17);
...
end PRODUCE;

task body CONSUME is
...
accept PUT(N:INTEGER) do
...
end PUT;
...
end CONSUME;
```

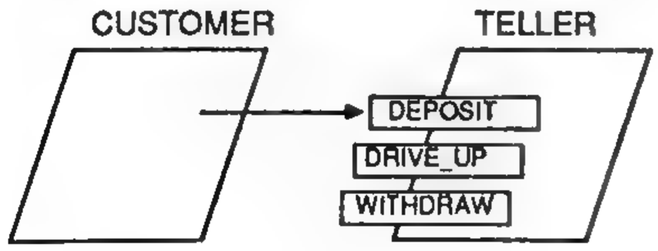

CLASSES OF RENDEZVOUS

- SIMPLE RENDEZVOUS
- OPTIONS FOR SERVING (CALLED) TASK

```
-- Simple selective wait
-- Selective wait with an else part
-- Selective wait with guards
-- Selective wait with delay alternative
-- Selective wait with terminate alternative
```

- OPTIONS FOR CALLING TASK

```
-- Conditional entry call
-- Timed entry call
```




```
task TELLER is

   entry DEPOSIT (ID : INTEGER; AMT : FLOAT);
   entry DRIVE_UP( . . .
   entry WITHDRAW ( . . .

end TELLER;
```


SIMPLE RENDEZVOUS

- the customer

```
TELLER.DEPOSIT ( ID  => 8064,
                 AMT => 100.0);
```

- the teller

```
...
accept DEPOSIT (ID : INTEGER; AMT : FLOAT) do
   ...
end DEPOSIT;
...
```

SELECTIVE WAIT

```
<selective_wait> ::=

    select
        <select_alternative>
    {or
        <select_alternative>}
    [else
        <sequence_of_statements>]
    end select;


<select_alternative> ::=

    [when <condition> =>]
     <selective_wait_alternative>


<selective_wait_alternative> ::=

        <accept_statement><sequence_of_statements>|
        <delay_alternative><sequence_of_statements>|
        terminate
```

- MUST CONTAIN AT LEAST ONE ACCEPT STATEMENT.
- CAN CONTAIN (mutually exclusively)
  -- one terminate alternative, or
  -- one or more delay alternatives, or
  -- an else part

SIMPLE SELECTIVE WAIT

- NONDETERMINISTICALLY SELECT ONE OF SEVERAL POSSIBLE ENTRIES.

```
...
loop
   select
      accept DEPOSIT (ID : INTEGER; AMT : FLOAT) do
         ...
      end DEPOSIT;
      ...
   or
      accept DRIVE_UP (ID : INTEGER; AMT : FLOAT) do
         ...
      end DRIVE_UP;
      ...
   or
      accept WITHDRAW (ID : INTEGER; AMT:out FLOAT) do
         ...
      end WITHDRAW;
      ...
   end select;
end loop;
...
```


SELECTIVE WAIT WITH ELSE OPTION

- IF NO ENTRIES PENDING, EXECUTE AN OPTIONAL SEQUENCE OF STATEMENTS.
- SERVING TASK DOES NOT GO INTO BLOCKED STATE.

```
loop
   select
      accept DEPOSIT (ID : INTEGER; AMT : FLOAT) do
         ...
      end DEPOSIT;
      ...
   or
      accept DRIVE_UP (ID : INTEGER; AMT : FLOAT) do
         ...
      end DRIVE_UP;
      ...
   else
      <sequence_of_statements>
   end select;
end loop;
...
```


ALTERNATIVES WITH GUARDS

- ALTERNATIVES WITHOUT GUARDS ARE ALWAYS OPEN.
- ALTERNATIVES WITH GUARDS THAT EVALUATE 'TRUE' ARE OPEN.
- ALTERNATIVES WITH GUARDS THAT EVALUATE 'FALSE' ARE CLOSED.
- IF ALL ALTERNATIVES ARE CLOSED AND THERE IS NO 'ELSE' PART, AN EXCEPTION IS RAISED.

```
loop
   select
      when BANKING_HOURS =>
         accept DEPOSIT (ID : INTEGER; AMT : FLOAT) do
            ...
         end DEPOSIT;
         ...
   or
      when DRIVE_UP_HOURS =>
         accept DRIVE_UP (ID : INTEGER; AMT : FLOAT) do
            ...
         end DRIVE_UP;
         ...
   end select;
end loop;
...
```

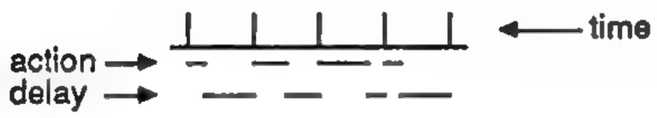
DELAY STATEMENT

- SUSPENDS FURTHER EXECUTION (OF THE TASK THAT EXECUTES THE DELAY) FOR AT LEAST THE DURATION SPECIFIED BY THE VALUE (IN SECONDS)

```
delay 10.0;
delay 0.0001;
```

- AN ALGORITHM FOR REPEATING AN ACTION EVERY SECOND:

```
declare
  INTERVAL   : constant := 1.0;
  TIME_HACK : CALENDAR.TIME  := CALENDAR.CLOCK;
begin
  loop
      delay DURATION (TIME_HACK - CALENDAR.CLOCK);

        -- action to be performed

    TIME_HACK := TIME_HACK + INTERVAL;
  end loop;
end;
```



PACKAGE CALENDAR

```
package CALENDAR is
 type TIME is private;

 subtype YEAR_NUMBER   is INTEGER range 1901 .. 2099;
 subtype MONTH_NUMBER is INTEGER range 1 .. 12;
 subtype DAY_NUMBER    is INTEGER range 1 .. 31;
 subtype DAY_DURATION  is DURATION range 0.0 .. 86_400.0;

 function CLOCK return TIME;

 function YEAR     (DATE : TIME) return YEAR_NUMBER;
 function MONTH    (DATE : TIME) return MONTH_NUMBER;
 function DAY      (DATE : TIME) return DAY_NUMBER;
 function SECONDS (DATE : TIME) return DAY_DURATION;

 procedure SPLIT (DATE      : in TIME;
                  YEAR      : out YEAR_NUMBER;
                  MONTH     : out MONTH_NUMBER;
                  DAY       : out DAY_NUMBER;
                  SECONDS   : out DAY_DURATION);

 function
      TIME_OF( YEAR      : YEAR_NUMBER;
               MONTH     : MONTH_NUMBER;
               DAY       : DAY_NUMBER;
               SECONDS : DAY_DURATION := 0.0) return TIME;

  function "+" (LEFT : TIME; RIGHT : DURATION) return TIME;
  function "+" (LEFT : DURATION; RIGHT : TIME) return TIME;
  function "-" (LEFT : TIME; RIGHT : DURATION) return TIME;
  function "-" (LEFT : TIME; RIGHT : TIME)     return DURATION;
  -- also functions for "<", "<=", ">", ">="
   TIME_ERROR : exception; -- raised by TIME_OF, "+" and "-"
private
   -- implementation-dependent
end CALENDAR;
```


SELECT WITH A DELAY ALTERNATIVE

- POSSIBLE RENDEZVOUS WITH THE CLOCK

```
loop
  select
     accept DEPOSIT (ID : INTEGER; AMT : FLOAT) do
       ...
     end DEPOSIT;
       ...

  or

     delay 10.0*MINUTES;
     <sequence_of_statements>

  end select;
end loop;
```


TERMINATE ALTERNATIVE

- CONSTITUTES AN 'OFFER' TO TERMINATE
- CONDITIONS FOR TERMINATION
  - -- Task master is completed
  - -- All dependent tasks (of master) are terminated or ready to terminate
  - -- No calling tasks in queue
  - -- i.e., If no task can ever again call this task

```
loop
  select
     accept DEPOSIT (ID : INTEGER; AMT : FLOAT) do
       ...
     end DEPOSIT;
       ...
  or

     terminate;

  end select;
end loop;
```

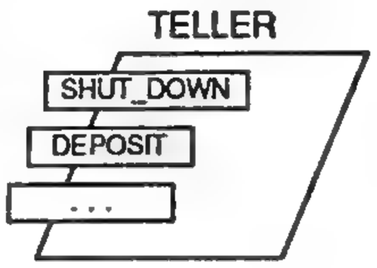
ABORT STATEMENT

- A TASK CAN ABORT ANY TASK WITHIN ITS VISIBILITY (INCLUDING ITSELF).
- RESULT IS UNCONDITIONAL TERMINATION.
- ALL DEPENDENT TASKS OF THE ABORTED TASK ARE ALSO ABORTED.

```
abort TELLER;
```

- OR, TO GIVE A TASK ITS LAST WISHES:

```
TELLER.SHUTDOWN;
delay 30.0;
abort TELLER;
```



TIMED ENTRY CALL

- CALLING TASK GETS INTO AN ENTRY QUEUE FOR A SPECIFIED MAXIMUM PERIOD OF TIME.
- CALLING TASK 'BALKS' THE QUEUE IF NOT SERVED WITHIN THAT AMOUNT OF TIME.

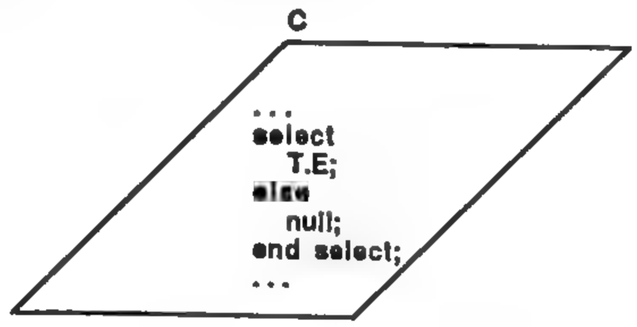
```
select
   TELLER.DEPOSIT (ID => 8064, AMT =>100.00);
or
   delay 30.0*MINUTES;
   DO_SOMETHING_ELSE;
end select;
```

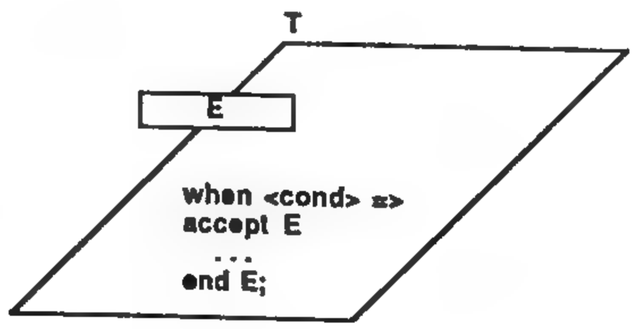

CONDITIONAL ENTRY CALL

- ATTEMPTS IMMEDIATE RENDEZVOUS
- ENTRY QUEUE IS EMPTY
- CALLED TASK IS ALREADY AT THE RENDEZVOUS POINT
- BEHAVES LIKE A TIMED ENTRY CALL WITH DELAY OF 0.0

```
select
  TELLER.WITHDRAW (ID => 8064, AMT => 1000.00);
else
  DO_SOMETHING_ELSE;
end select;
```


TIMED ENTRY CALLS AND CONDITIONAL ENTRY CALLS CAN BE USED TO CALL ENTRIES WHICH ARE GUARDED





APPLICATIONS FOR TASKS

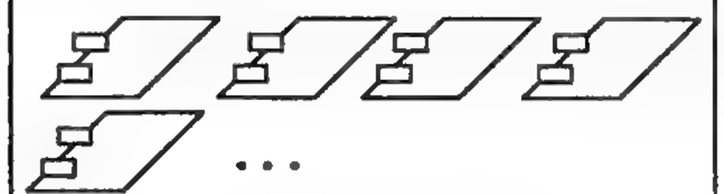
- CONCURRENT OPERATIONS
- MESSAGE ROUTING
- SHARED RESOURCE MANAGEMENT
- INTERRUPT HANDLING

MATRIX MULTIPLICATION

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 0 \end{bmatrix} * \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

```
type ROW_OR_COL is array (INTEGER range <>) of INTEGER;
type PTR is access ROW_OR_COL;

task type PARTIAL is
    entry SEND  (ROW, COL : ROW_OR_COL);
    entry RECEIVE (RESULT : out INTEGER);
end PARTIAL;
```

MAIN



```
begin
  -- send row and col
  -- receive partial product
end
```



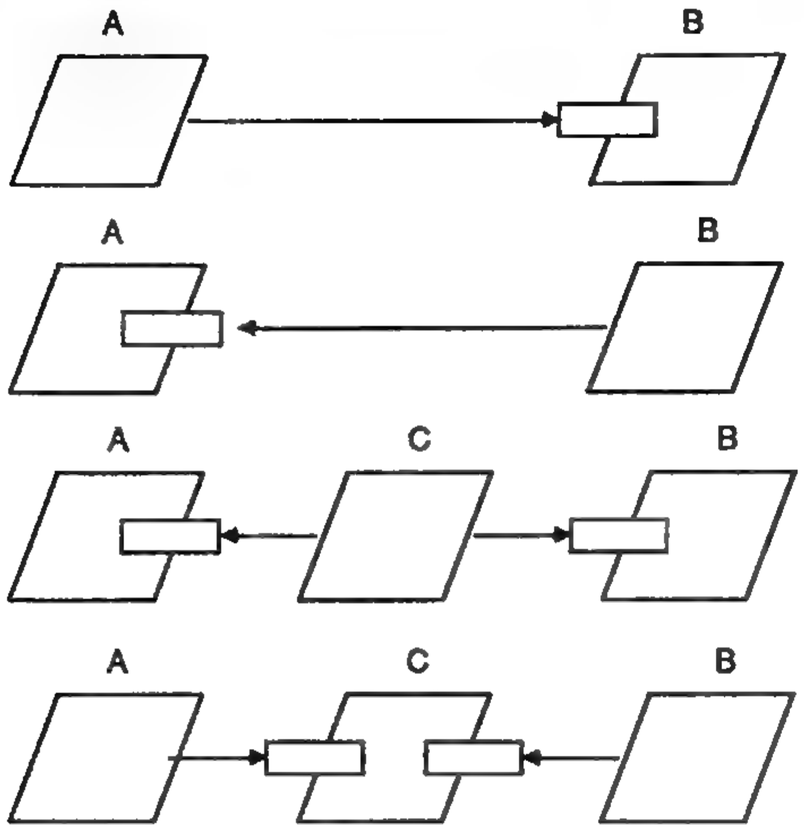
```
task body PARTIAL is

    PRODUCT  : INTEGER := 0;
    ROW_PTR  : PTR;
    COL_PTR  : PTR;

begin
    accept SEND (ROW, COL : ROW_OR_COL) do
        ROW_PTR := new ROW_OR_COL'(ROW);
        COL_PTR  := new ROW_OR_COL'(COL);
    end SEND;

    for J in ROW_PTR.all'RANGE
    loop
        PRODUCT := PRODUCT +
            ROW_PTR(J) * COL_PTR(J);
    end loop;

    accept RECEIVE (RESULT : out INTEGER) do
        RESULT := PRODUCT;
    end RECEIVE;
end PARTIAL;
```



```
procedure MAIN is

  COLS  : constant := 10;
  ROWS : constant := 10;
  type MATRIX is array (1 .. ROWS) of
                          ROW_OR_COL (1 .. COLS);

    MAT     : MATRIX;
    VECTOR : ROW_OR_COL (1 .. COLS);
    FINAL    : ROW_OR_COL (1 .. ROWS);
begin
  ...
  declare

    WORKER : array (1 .. ROWS) of PARTIAL;  -- tasks

  begin
    for J in 1 .. ROWS
    loop
        WORKER(J).SEND(ROW => MAT(J),
                                COL => VECTOR);
    end loop;

    for J in 1 .. ROWS
    loop
        WORKER(J).RECEIVE (FINAL(J));
    end loop;

  end;       -- block
  ...

end MAIN;
```

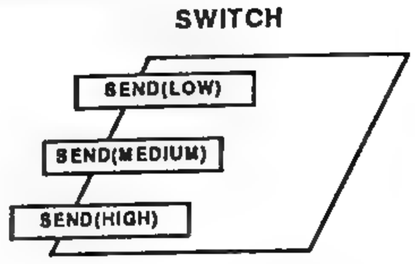
MESSAGE ROUTING

TO SEND A MESSAGE FROM TASK A TO TASK B



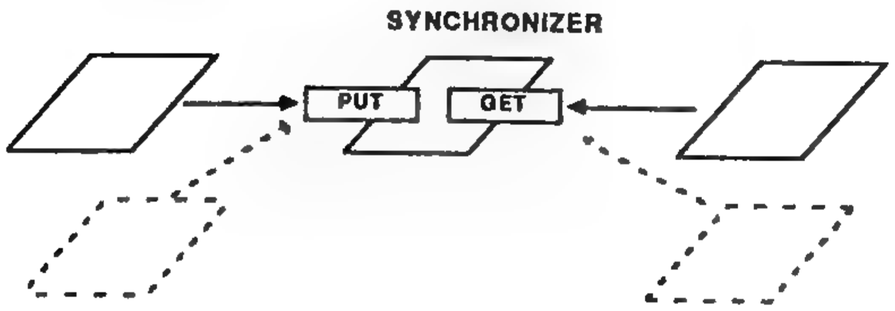
PRIORITY MESSAGES

```
type PRIORITY is (LOW, MEDIUM, HIGH);

task SWITCH is
   entry SEND (PRIORITY)
         (M : in STRING);
end SWITCH;

task body SWITCH is
begin
   loop
      select

         accept SEND(HIGH) (M : in STRING) do . . . end SEND;

      or

         when SEND(HIGH)'COUNT = 0 =>
         accept SEND(MEDIUM) (M : in STRING) do  . . . end SEND;

      or

         when SEND(HIGH)'COUNT = 0 and
              SEND(MEDIUM)'COUNT = 0 =>
         accept SEND (LOW)(M : in STRING) do . . . end SEND;

      end select;
   end loop;
end SWITCH;
```



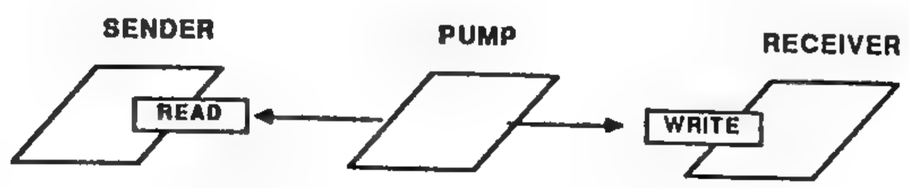
A SYNCHRONIZING BUFFER

```
task SYNCHRONIZER is
   entry PUT (ITEM : in SOME_TYPE);
   entry GET (ITEM : out SOME_TYPE);
end  SYNCHRONIZER;

task body SYNCHRONIZER is

   SPOT : SOME_TYPE;

begin

   loop

      accept PUT (ITEM : in SOME_TYPE) do
         SPOT := ITEM;
      end  PUT;

      accept GET (ITEM : out SOME_TYPE) do
         ITEM := SPOT;
      end GET;

   end loop;

end  SYNCHRONIZER;
```



PUMPING TASK

```
task PUMP;

task SENDER is
   entry READ (ITEM : out SOME_TYPE);
end SENDER;

task RECEIVER is
   entry WRITE (ITEM : in SOME_TYPE);
end  RECEIVER;

task body PUMP is
   THE_ITEM : SOME_TYPE;
begin
   loop
      SENDER.READ (THE_ITEM);
      RECEIVER.WRITE (THE_ITEM);
   end loop;
end PUMP;

task body SENDER    is separate;
task body RECEIVER  is separate;
```



CONTROLLING RESOURCES

SEVERAL CONCERNS ARE PRESENT WHEN DEALING WITH PARALLELISM THAT ARE NOT PRESENT WHEN DEALING IN A PURELY SEQUENTIAL MODE

IT IS IMPORTANT TO BE ABLE TO ASSURE THAT A VALUE IS NOT BEING CHANGED BY ONE USER AT THE PRECISE MOMENT THAT IT IS BEING REFERENCED BY ANOTHER USER

- Ada PROVIDES A PRAGMA 'SHARED' WHICH CAN HELP

```
INDEX : Integer;
 pragma SHARED(INDEX);
```

- ENFORCES MUTUALLY EXCLUSIVE ACCESS
- AVAILABLE FOR SCALAR AND ACCESS TYPES ONLY

SEMAPHORES

```
task SEMAPHORE is

    entry SEIZE;
    entry RELEASE;

end SEMAPHORE;


task body SEMAPHORE is

begin
  loop

        accept SEIZE;

        accept RELEASE;

  end loop;
end SEMAPHORE;
```


ENCAPSULATING A DATA ITEM

```
task PROTECTED is
      entry SET (OBJ : in  SOME_TYPE);
      entry GET (OBJ : out SOME_TYPE);
end  PROTECTED;

task body PROTECTED is
      LOCAL : SOME_TYPE;
begin
   accept SET (OBJ : in SOME_TYPE) do
       LOCAL := OBJ;
   end SET;
  loop
     select
          accept SET (OBJ : in SOME_TYPE) do
              LOCAL := OBJ;
          end SET;
     or
          accept GET (OBJ : out Integer) do
              OBJ := LOCAL;
          end GET;
     end select;
  end loop;
end  PROTECTED;
```


HARDWARE INTERRUPTS

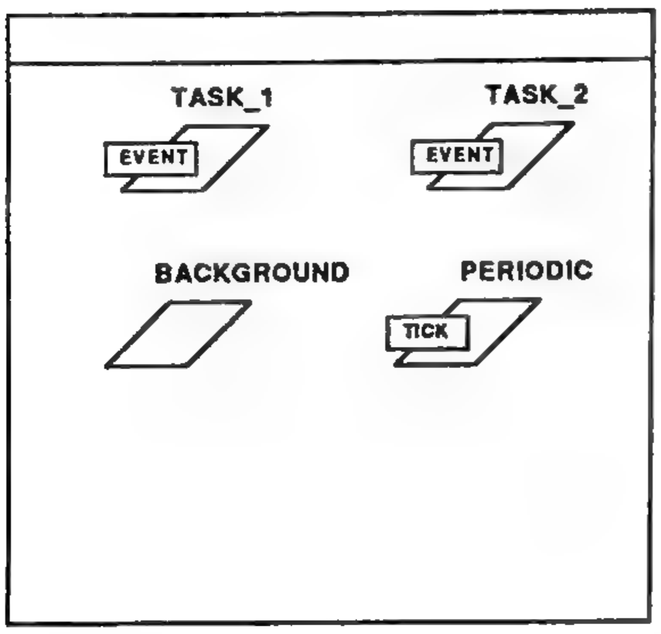
- FOR ARCHITECTURES THAT 'JUMP' TO A CERTAIN HARDWARE ADDRESS UPON RECEIPT OF AN INTERRUPT
- A TASK ENTRY IS ASSOCIATED WITH THE ADDRESS
- PRIORITY IS HIGHER THAN ANY USER-DEFINED

```
task INTERRUPT_HANDLER is
   entry DONE;
   for DONE use at 16#40#;
end  INTERRUPT_HANDLER;


task body INTERRUPT_HANDLER is
begin
  loop
     accept DONE do
     ...
     end DONE;
  end loop;
end  INTERRUPT_HANDLER;
```

EVENT DRIVEN SYSTEMS W/BACKGROUND

- A cyclic executive might deal with several levels of processing
  - Event driven processing (high priority, perhaps interrupt handling)
  - Periodic (cyclic) processing
  - Background processing (low priority)

EXECUTIVE




```
procedure EXECUTIVE is

   task TASK_1 is
      pragma PRIORITY (10);
      entry EVENT;
   end TASK_1;

   task TASK_2 is
      entry EVENT;
      for EVENT use at 16#110#;
   end TASK_2;

   task BACKGROUND is
      pragma PRIORITY (0);
   end BACKGROUND;

   task PERIODIC is
      pragma PRIORITY (5);
      entry TICK;                -- one tick per cycle
   end PERIODIC;

   task body PERIODIC is
      ...
   begin
      loop
         accept TICK;
         ... -- process a frame
      end loop;
   end PERIODIC;

      -- bodies (or stubs) of other tasks go here

end EXECUTIVE;
```


EXCEPTIONS

- WHEN AN EXCEPTION IS RAISED, EXECUTION IS ABANDONED AND AN EXCEPTION HANDLER IS SOUGHT
- PREDEFINED EXCEPTIONS
  - CONSTRAINT_ERROR
    raised when a range, index, or discriminant constraint is violated
  - NUMERIC_ERROR
    raised when a numeric operation yields a result that cannot be represented
  - PROGRAM_ERROR
    raised when all alternatives of a select statement having no else part are closed or if an erroneous condition is detected
  - STORAGE_ERROR
    raised when insufficient storage remains for a given collection of designated objects
  - TASKING_ERROR
    raised by trying to communicate with a dead task

USER-DEFINED EXCEPTIONS

- BASIC DECLARATIVE ITEMS
- CAN ONLY BE RAISED EXPLICITLY

```
UNDER_FLOW, OVER_TEMP : exception;

raise UNDER_FLOW;

raise NUMERIC_ERROR;

raise;
```

SUPPRESSION OF CHECKS

- RUNTIME CHECKS IMPOSE A CERTAIN OVERHEAD
- CHECKS CAN BE TURNED OFF
- EFFECTS OF TURNING OFF CHECKS CAN BE LIMITED TO CERTAIN OBJECTS AND CERTAIN UNITS
- CHECKS THAT RAISE PREDEFINED EXCEPTIONS
  - access_check, discriminant_check, index_check,
  - length_check, range_check, division_check,
  - overflow_check, elaboration_check, storage_check
- SETTING THE CHECK-SUPPRESSION

```
pragma SUPPRESS (index_check, ON => MY_INDEX);
```

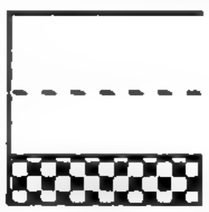

EXCEPTION HANDLERS

- CAN APPEAR AT THE END OF A BLOCK STATEMENT, SUBPROGRAM, PACKAGE OR TASK
- TAKE THE FORM OF A CASE STATEMENT
- CAN CONTAIN AN 'OTHERS' HANDLER
- EXCEPTIONS NOT HANDLED IN THE 'NEAREST' HANDLER ARE PROPAGATED

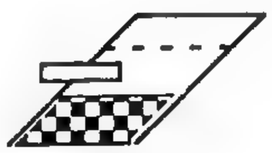
FRAMES OF REFERENCE

SUBPROGRAM BODY

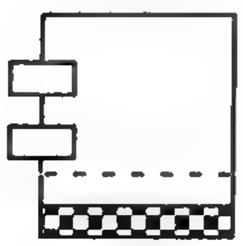
BLOCK STATEMENT



TASK BODY



PACKAGE BODY



EXCEPTIONS RAISED IN BLOCKS

EXCEPTION HANDLER EXISTS

```
-- Exception is handled and control passes
-- to the next sequential statement following the
-- block statement
```

NO EXCEPTION HANDLER EXISTS

```
-- Exception is propagated statically (the same error
-- is raised at the next sequential statement following
-- the block statement)
```

EXCEPTION IS RAISED IN DECLARATIVE PART

```
-- Exception is immediately raised at the next
-- sequential statement following the block statement
```

EXCEPTIONS RAISED IN SUBPROGRAMS

EXCEPTION HANDLER EXISTS

-- Exception is handled and control passes to the
-- point of call

NO EXCEPTION HANDLER EXISTS

-- Exception is propagated dynamically (the same
-- error is raised at the point of call)

EXCEPTION IS RAISED IN DECLARATIVE PART

-- Exception is immediately raised at the point
-- of call of the subprogram

EXCEPTIONS RAISED IN PACKAGES

- PACKAGE IS A DECLARATIVE ITEM (NESTED)

EXCEPTION HANDLER EXISTS

-- Exception is handled and elaboration of the package
-- body is completed

NO EXCEPTION HANDLER EXISTS

-- The same exception is raised following the
-- declarative item

EXCEPTION IS RAISED IN DECLARATIVE PART

-- The same exception is raised following the
-- declarative item

- PACKAGE IS A COMPILATION UNIT

EXCEPTION HANDLER EXISTS

-- Exception is handled and elaboration is complete

ALL OTHER CASES

-- Execution of main program is abandoned

EXCEPTIONS RAISED IN TASKS

EXCEPTION HANDLER EXISTS

-- Exception is handled and the task is complete

NO EXCEPTION HANDLER EXISTS

-- Task is complete

EXCEPTION IS RAISED IN DECLARATIVE PART

-- Task is complete and the tasking_error exception
-- is raised at the point of activation of task

- EXCEPTIONS RAISED DURING TASK COMMUNICATION

-- A tasking_error is raised in the calling task if
-- called task is completed before rendezvous takes
-- place

-- When an exception is raised in the called task,
-- the same error is propagated to the calling task

-- When an exception is raised in the calling task,
-- the same error is not propagated to the called task

- AN ANONYMOUS RAISE STATEMENT ALLOWS
  PARTIAL HANDLING WITH MORE COMPLETE HANDLING
  ACCOMPLISHED AT AN OUTER LEVEL

```
exception
   when numeric_error =>
      <sequence_of_statements>
      raise;   -- same exception is propagated
end;
```

- YOU CAN PROPAGATE AN EXCEPTION BEYOND ITS SCOPE

```
begin
   ...
   declare
      LOCAL_EXCEPTION : exception;
   begin
      ...
      raise LOCAL_EXCEPTION;
   end;  -- no exception handler
   ...
exception
   when others =>
      <sequence_of_statements>
end;
```

REPRESENTATION SPECIFICATIONS

- ALLOW THE USER TO GET DOWN TO THE BIT LEVEL OF THE UNDERLYING ARCHITECTURE
- PROVIDE MACHINE-DEPENDENT CAPABILITY
- ARE NOT PART OF THE ACVC
- THE USER CAN SPECIFY:
  - SIZE
  - RECORD TYPE REPRESENTATION
  - ENUMERATION TYPE REPRESENTATION
  - ADDRESS SPECIFICATION

SIZE REPRESENTATION

- TO DICTATE SIZE OF OBJECTS OF A TYPE

```
type MY_RANGE is  range -100 .. 100;
for MY_RANGE'SIZE use 8;  -- bits
```

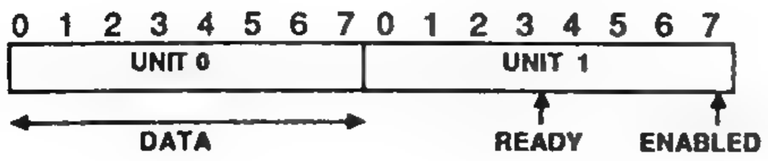
- TO DICTATE SIZE OF A COLLECTION OF DESIGNATED OBJECTS

```
BYTES : constant := 8;   -- bits

type SOME_TYPE is . . .
type PTR is access SOME_TYPE;

for PTR'STORAGE_SIZE use 1000*BYTES;
```


RECORD TYPE REPRESENTATION

- SPECIFIES ORDER, POSITION, SIZE OF COMPONENTS
- SPECIFIES MULTIPLE UNIT ALLIGNMENT

```
type IO_PORT is
  record
    DATA      : INTEGER range 0 .. 255;
    READY     : BOOLEAN;
    ENABLED   : BOOLEAN;
  end record;

for IO_PORT use
  record at mod 2;    -- double unit boundary
    DATA      at 0 range 0 .. 7;
    READY     at 1 range 3 .. 3;
    ENABLED   at 1 range 7 .. 7;
  end record;
```



ENUMERATION TYPE REPRESENTATION

- ALLOWS THE USER TO DICTATE THE UNDERLYING REPRESENTATION OF LITERALS OF AN ENUMERATED TYPE
- NUMERIC ORDER MUST NOT VIOLATE PREDEFINED ORDER
- SUCC, PRED, POS ARE DEFINED EVEN WHEN GAPS EXIST IN UNDERLYING REPRESENTATION

```
type RESPONSE is (UP, DOWN, LEFT, RIGHT);

for RESPONSE'SIZE use 4;

for RESPONSE use ( UP    => 2#0001#,
                   DOWN  => 2#0010#,
                   LEFT  => 2#0100#,
                   RIGHT => 2#1000#);
```

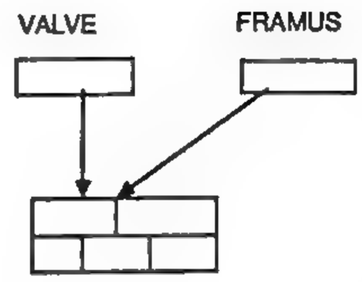
ADDRESS REPRESENTATION

- ALLOWS THE USER TO DICTATE THE ACTUAL ADDRESS OF OBJECTS, SUBPROGRAMS AND TASKS

```
COUNTER : INTEGER;
for COUNTER use at 16#100#;

procedure EMERGENCY;
for EMERGENCY use at 16#FF4E#;

task MONITOR is
   entry FAILURE;
   for FAILURE use at 8#7776#;
end MONITOR;
```


CAVEAT EMPTOR

```
generic
   type OBJECT is limited private;
   type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);
...
type MY_TYPE is ...
type POINTER is access MY_TYPE;

procedure FREE is new UNCHECKED_DEALLOCATION
                      (OBJECT => MY_TYPE,
                       NAME  => POINTER);

VALVE, FRAMUS : POINTER;
...
VALVE := new MY_TYPE;
FRAMUS := VALVE;
...
FREE (VALVE);
```



CAVEAT EMPTOR

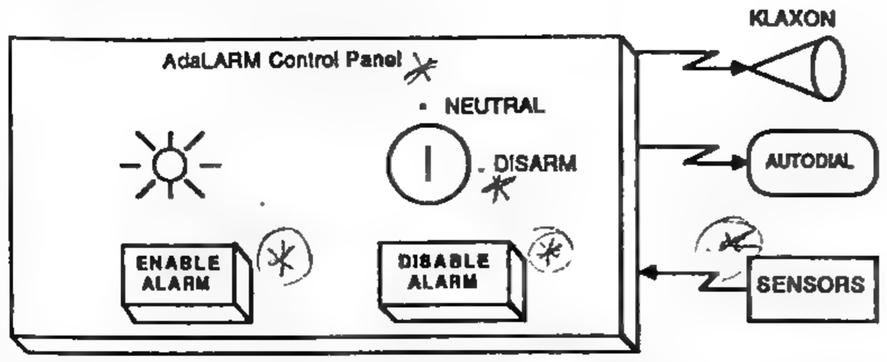
```
generic

   type SOURCE is limited private;
   type TARGET is limited private;

function UNCHECKED_CONVERSION(S : SOURCE)
                              return TARGET;
```

- Returns the (uninterpreted) parameter value as a value of the target type.
- Usually generates no additional code
- It is the programmers responsibility to ensure that conversion maintains the properties of the target type

OTHER LANGUAGES

- A SUBPROGRAM WRITTEN IN ANOTHER LANGUAGE CAN BE CALLED FROM AN ADA PROGRAM
- ALL COMMUNICATION MUST BE ACHIEVED VIA PARAMETERS AND FUNCTION RESULTS
- A PRAGMA MUST BE GIVEN FOR EACH SUBPROGRAM
- SUBPROGRAM BODY IS NOT ALLOWED
- CAPABILITY NEED NOT BE PROVIDED BY AN IMPLEMENTATION

```
package FORT_LIB IS

   function SQRT (X : FLOAT) return FLOAT;
   function EXP  (X : FLOAT) return FLOAT;

private

   pragma INTERFACE (FORTRAN, SQRT);
   pragma INTERFACE (FORTRAN, EXP);

end FORT_LIB;
```



- **ENABLING: When the Enable Alarm button is pressed, the indicator light goes on and the sensors are activated after approximately 1 minute. The key indicator must be at 'neutral'. The enable button has no affect if the light is not 'off'.**
- **DISABLING: When the Disable Alarm button is pressed, the indicator light goes off and the sensors are immediately deactivated. The disable button has no affect if the light is not 'on' (steady).**
- **ARMING: If the alarm is enabled and a sensor detects an intruder, the alarm becomes armed (the indicator light begins to blink). If the alarm is not disarmed (see below) within 1 minute, the klaxon is sounded and the security office is automatically dialed.**
- **DISARMING: The alarm is disarmed by inserting the key and turning it clockwise (to 'disarm'). When this is done, the light and the klaxon are turned off but the owner must call the security office personally. The key must be turned counterclockwise (to 'neutral') before the alarm can again be enabled.**

AdaLARM Project

Design an implementation for the AdaLARM system subject to the following conditions:

- The AdaLARM system uses an eight-bit processor with certain devices memory mapped. All hardware interrupts cause a vector to octal location 40. A status word is located at octal location 42 and gives additional information about the interrupts:

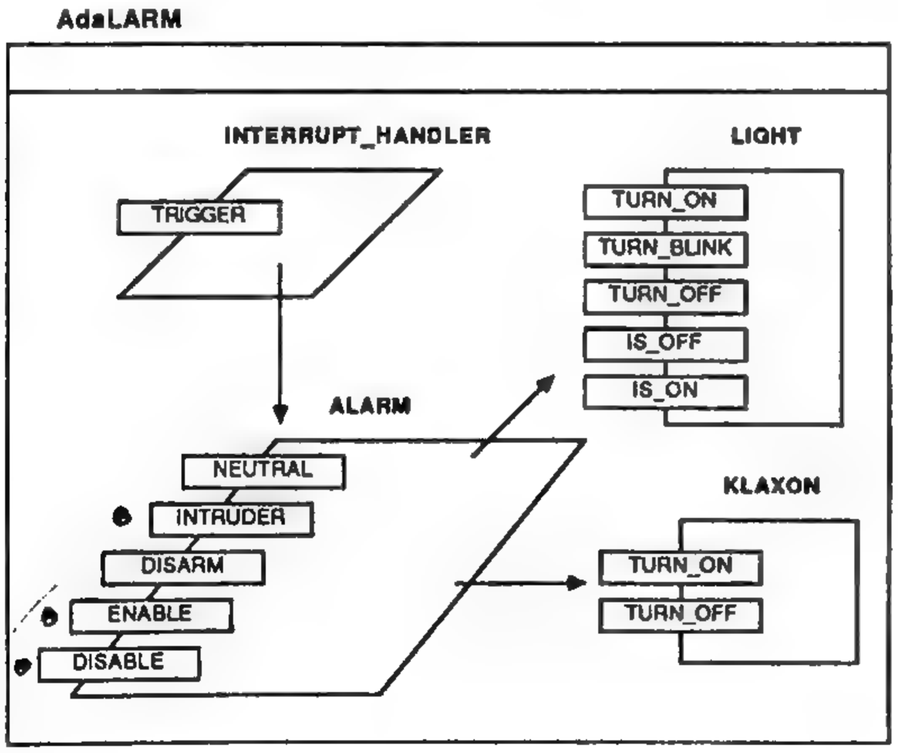
INTERRUPT	STATUS WORD
Enable button	00000001
Disable button	00000010
Key to 'disarm'	00000100
Key to 'neutral'	00010000
Sensor trigger	00001000

- The autodial to the security office takes place automatically when the klaxon is sounded.
- The light is mapped to octal location 50 and has the following representation:

LIGHT STATUS	REPRESENTATION
Light is off	00000000
Light is on (steady)	11111111
Light is blinking	00001111

- The Klaxon is mapped to octal location 60 and has the following representation:

KLAXON STATUS	REPRESENTATION
Klaxon is sounding	11111111
Klaxon is silent	00000000




```
procedure AdaLARM is

    task INTERRUPT_HANDLER is
        entry TRIGGER;
        for TRIGGER use at 8#40#;
    end INTERRUPT_HANDLER;

    task ALARM is
        entry NEUTRAL;
        entry INTRUDER;
        entry DISARM;
        entry ENABLE;
        entry DISABLE;
    end ALARM;

    package KLAXON is
        procedure TURN_ON;
        procedure TURN_OFF;
    end KLAXON;

    package LIGHT is
        procedure TURN_ON;
        procedure TURN_BLINK;
        procedure TURN_OFF;
        function IS_ON  return BOOLEAN;
        function IS_OFF return BOOLEAN;
    end LIGHT;

    task body INTERRUPT_HANDLER is separate;
    task body ALARM             is separate;
    package body KLAXON         is separate;
    package body LIGHT          is separate;

begin
    null;    -- let the tasks do all the work
end AdaLARM;
```

```
separate (AdaLARM)
package body LIGHT is
    type LIGHT_STATUS is (OFF, BLINK, ON);
    for LIGHT_STATUS'SIZE use 8;  -- bits

    for LIGHT_STATUS use ( OFF    => 2#00000000#,
                           BLINK  => 2#00001111#,
                           ON     => 2#11111111#);

    BULB : LIGHT_STATUS := OFF;
    for BULB use at 8#50#;

    procedure TURN_ON is
    begin
        BULB := ON;
    end;

    procedure TURN_BLINK is
    begin
        BULB := BLINK;
    end;

    procedure TURN_OFF is
    begin
        BULB := OFF;
    end;

    function IS_ON return BOOLEAN is
    begin
        return BULB = ON;
    end;

    function IS_OFF return BOOLEAN is
    begin
        return BULB = OFF;
    end;
end LIGHT;
```



```
separate (AdaLARM)
package body KLAXON is

    task SECURITY_OFFICE is
        entry CALL;
    end;

    type KLAXON_STATUS is (OFF, ON);
    for KLAXON_STATUS'SIZE use 8;

    for KLAXON_STATUS use (OFF  => 2#00000000#,
                           ON   => 2#11111111#);

    HORN : KLAXON_STATUS := OFF;
    for HORN use at 8#60#;

    task body SECURITY_OFFICE is separate;

    procedure TURN_ON is
    begin
        HORN := ON;
        SECURITY_OFFICE.CALL;
    end;

    procedure TURN_OFF is
    begin
        HORN := OFF;
    end;

end KLAXON;
```



```
separate (AdaLARM)
task body INTERRUPT_HANDLER is

    type STATUS is ( ENABLE, DISABLE, KEY_DISARM,
                     SENSOR, KEY_NEUTRAL);

    for STATUS'SIZE use 8;

    for STATUS use ( ENABLE       => 2#00000001#,
                     DISABLE      => 2#00000010#,
                     KEY_DISARM   => 2#00000100#,
                     SENSOR       => 2#00001000#,
                     KEY_NEUTRAL  => 2#00010000#);

    STATUS_WORD : STATUS;

    for STATUS_WORD use at 8#42#;

    WORD : STATUS;      -- Saves the STATUS_WORD to
                        -- avoid the 'simultaneous'
                        -- interrupt problem.
```



```
begin  -- INTERRUPT_HANDLER;
  loop
    accept TRIGGER do
        WORD := STATUS_WORD;
    end TRIGGER;

    -- Perhaps an exception handler in case of
    -- multiple interrupts

    case WORD is

        when ENABLE      =>  select
                                 ALARM.ENABLE;
                             else
                                 null;
                             end select;

        when DISABLE     =>  select
                                 ALARM.DISABLE
                             else
                                 null;
                             end select;

        when KEY_DISARM  =>  ALARM.DISARM;

        when KEY_NEUTRAL =>  ALARM.NEUTRAL

        when SENSOR      =>  select
                                 ALARM.INTRUDER;
                             else
                                 null;
                             end select;

    end case;
  end loop;
end INTERRUPT_HANDLER;
```

```
separate (AdaLARM)
task body ALARM is

        type KEY_TYPE is (NEUTRAL_STATE,
                          DISARM_STATE);

        KEY : KEY_TYPE := NEUTRAL_STATE;
begin
    loop

        select
(1)         accept NEUTRAL;
            LIGHT.TURN_OFF;
            KEY := NEUTRAL_STATE;

        or
(3)         when KEY = NEUTRAL_STATE and
                                LIGHT.IS_OFF  =>
            accept ENABLE;
            LIGHT.TURN_ON;
            delay 60.0;

        or
(2)         when LIGHT.IS_ON  =>

            accept DISABLE;
            LIGHT.TURN_OFF;

    or
```



```
    -- or
(4)     when LIGHT.IS_ON  =>
            accept INTRUDER;
            LIGHT.TURN_BLINK;

            select
                accept DISARM;          -- wait for deactivation
                LIGHT.TURN_OFF;         -- klaxon is not sounding
                KEY := DISARM_STATE;
            or
                delay 60.0              -- allow time to insert key
                KLAXON.TURN_ON;

            end select;

    or

            accept DISARM;
            KLAXON.TURN_OFF;
            LIGHT.TURN_OFF;             -- klaxon is sounding or key
            KEY := DISARM_STATE;        -- simply turned to disarm by
                                        -- mistake

    end select;

    end loop;
end ALARM;
```

PROGRAM STRUCTURE

- A program is a collection of one or more compilation units submitted to a compiler in one or more compilations
- The compilation units of a program are said to belong to a program library
- A compilation unit defines either a library unit or a secondary unit

<compilation> ::= {<compilation_unit>}

<compilation_unit> ::=
    <context_clause><library_unit> |
    <context_clause><secondary_unit>

<context_clause> ::=
    {with_clause {use_clause}}

LIBRARY UNITS

- SUBPROGRAM DECLARATION (SPECIFICATION)
- PACKAGE DECLARATION (SPECIFICATION)
- GENERIC DECLARATION (SPECIFICATION)
- SUBPROGRAM BODY (only if there is no distinct subprogram declaration as a library unit)
- GENERIC INSTANTIATION

SECONDARY UNITS

- LIBRARY UNIT BODY
  - -- SUBPROGRAM BODY
  - -- PACKAGE BODY
- SUBUNIT

NOTE: A 'WITH' CLAUSE ALWAYS REFERS TO A LIBRARY UNIT, NEVER TO A SECONDARY UNIT

SUBUNITS

- A body stub is only allowed as the body of a program unit (a subprogram, package, task or generic unit) if the body stub occurs immediately within the declarative part of another compilation unit.
- Visibility within the subunit is the visibility that would be obtained at the place of the corresponding body stub (within the parent unit) if the with clauses and use clauses of the subunit were appended to the context clause of the parent unit.
- The simple names of all subunits that have the same ancestor library unit must be distinct identifiers.
- An operator symbol cannot be the designator of a subunit.

ORDER OF COMPILATION

1. A compilation unit must be (re)compiled after all library units named by its context clause.

2. A secondary unit that is a subprogram or package body must be (re)compiled after the corresponding library unit.

3. Any subunit of a parent compilation unit must be (re)compiled after the parent compilation unit

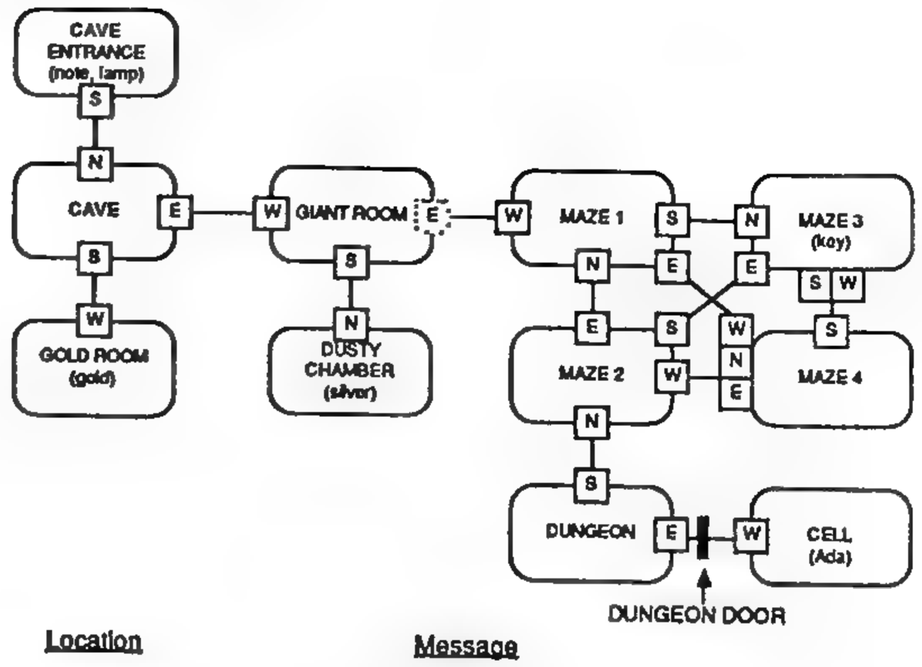
FANTASY SIMULATION GAMES

In games like ADVENTURE and ZORK (DUNGEON) the adventurer enters commands which are subsequently executed. If the player enters words which are not part of the vocabulary of the game, an error message will be generated and the player will be able to attempt another command. If the command is valid (contains only words from the vocabulary in their expected grammatical order) but the command has no valid meaning (GO KNIFE), then a different error is generated and the player again gets another chance. Commands in such games move the player from place to place, allow the player to pick up and drop items, allow the player to inventory his current holding of items etc.

The game we will implement has a limited map (11 locations) and a very limited vocabulary (GO, TAKE, DROP, OPEN, LIGHT, UNLOCK, READ, SAY, INVENTORY, QUIT, NORTH, EAST, WEST, SOUTH, LAMP, KEY, DOOR, ADA, GOLD). A valid command is of the form VERB- NOUN such as OPEN DOOR, GO NORTH etc.

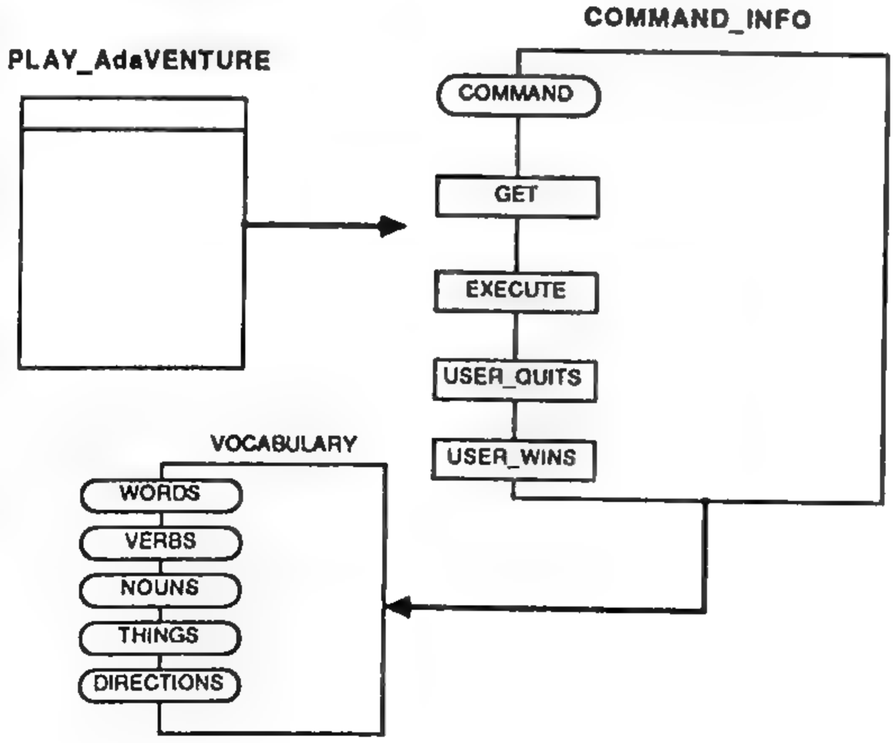
The game must keep track of such state information as player's location and current inventory as well as the current inventory of each location. The goal is to rescue Ada from the locked cell, find the gold and silver, and escape to the cave entrance. There is a door which separates the dungeon from the cell. Once unlocked, it remains unlocked, once open, it remains open.

There is a secret passage in the Giant's room which opens into the maze only if the player has uttered the magic word ("abracADAbra") while in the giant's room.



Location	Message
ENTRANCE	"You are at the entrance of a cave"
CAVE	"You are in a large cave"
GOLD_ROOM	"You have entered the gold room"
GIANT_ROOM	"You are in the giant's room"
CHAMBER	"You've entered a dusty chamber, a sign says 'abracADAbra'"
all mazes	"You are in a maze of twisty passages all alike"
DUNGEON	"You have found the dungeon"
CELL	"You are in a damp cell"

The main program will GET COMMANDs from the player and will EXECUTE these COMMANDs. This process will continue until either the PLAYER QUITS or the PLAYER WINS. COMMANDs will be represented as VERB-NOUN pairs from some VOCABULARY.




```
package VOCABULARY is

    type WORDS is ( NORTH, EAST, WEST, SOUTH,
                    GOLD, SILVER, NOTE, LAMP, KEY,
                    Ada, MAGIC_WORD, DOOR, GO,
                    TAKE, LIGHT, DROP, READ, SAY,
                    OPEN, UNLOCK, QUIT, INVENTORY);

    subtype NOUNS      is WORDS range NORTH .. DOOR;
    subtype VERBS      is WORDS range GO .. INVENTORY;

    subtype DIRECTIONS is NOUNS range NORTH .. SOUTH;

        -- Primarily used with the GO verb.

    subtype THINGS     is NOUNS range GOLD .. Ada;

        -- These are THINGS that can be carried by the player
        -- and that can be found in various locations.

end VOCABULARY;
```

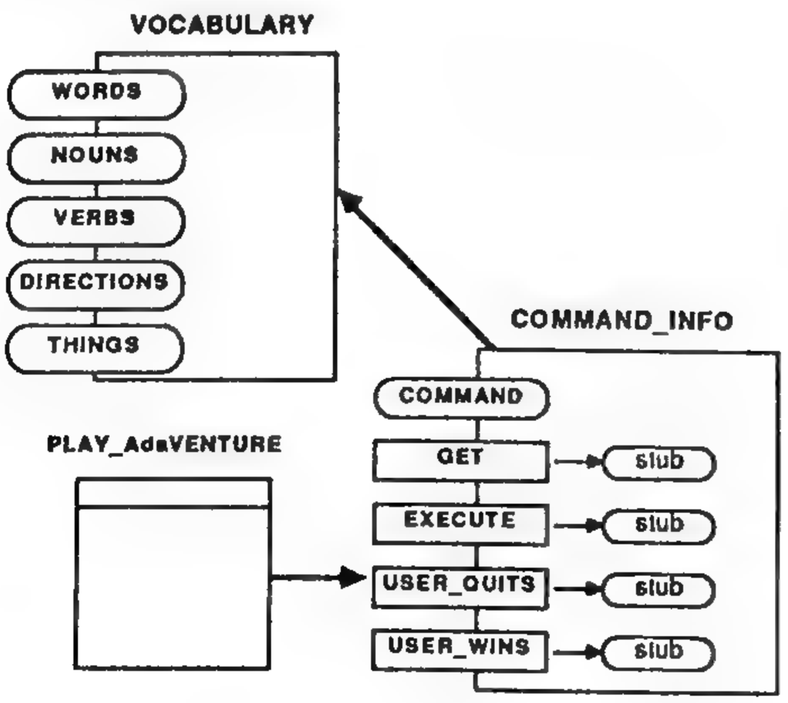
```
with VOCABULARY;
package COMMAND_INFO is

    type COMMAND is private;

    procedure GET (C : out COMMAND);

        -- This procedure interacts with the player to get a
        -- legal command.  If the command is not legal, the
        -- GET routine will continue to interrogate the player
        -- until a legal command is finally entered.

    procedure EXECUTE (C : in COMMAND);

        -- This procedure performs the action indicated
        -- by the player.  Silly (legal but invalid) commands such
        -- as 'GO KEY' are  treated with the respect they deserve.
        -- Valid commands are carried out.

    function   USER_QUITS (C : COMMAND)  return boolean;

    function   USER_WINS  return boolean;

private

    type COMMAND is
        record

            VERB  : VOCABULARY.VERBS;

            NOUN : VOCABULARY.NOUNS;

        end record;

end COMMAND_INFO;
```



```
with COMMAND_INFO; use COMMAND_INFO;
procedure PLAY_AdaVENTURE is

   THE_COMMAND : COMMAND_INFO.COMMAND;

begin

    loop

        GET (THE_COMMAND);

        EXECUTE (THE_COMMAND);

        exit when USER_QUITS (THE_COMMAND)
             or     USER_WINS;

    end loop;

        -- some final message could be printed here.

end PLAY_AdaVENTURE;
```



```
package body COMMAND_INFO is

    procedure GET (C : out COMMAND)          is separate;

    function USER_QUITS (C : COMMAND)
                  return BOOLEAN             is separate;

    function USER_WINS return BOOLEAN        is separate;

    procedure EXECUTE (C : in COMMAND)       is separate;

end COMMAND_INFO;
```

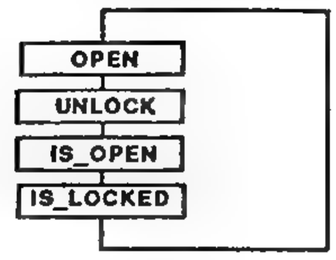



```
with TEXT_IO;
separate  (COMMAND_INFO)
procedure EXECUTE (C : in COMMAND) is      -- a subunit

        procedure GO_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure DROP_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure TAKE_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure OPEN_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure UNLOCK_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure READ_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure SAY_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure LIGHT_RTN (NOUN : in VOCABULARY.NOUNS)
                          is separate;
        procedure INVENTORY_RTN is separate;

use VOCABULARY;   -- for direct visibility
begin

    case C.VERB is
       when GO          => GO_RTN       (NOUN => C.NOUN);
       when TAKE        => TAKE_RTN     (NOUN => C.NOUN);
       when DROP        => DROP_RTN     (NOUN => C.NOUN);
       when OPEN        => OPEN_RTN     (NOUN => C.NOUN);
       when UNLOCK      => UNLOCK_RTN   (NOUN => C.NOUN);
       when LIGHT       => LIGHT_RTN    (NOUN => C.NOUN);
       when INVENTORY   => INVENTORY_RTN;
       when SAY         => SAY_RTN      (NOUN => C.NOUN);
       when READ        => READ_RTN     (NOUN => C.NOUN);
       when OTHERS      => null;
    end case;

end  EXECUTE;
```

Ada as Pseudo Code

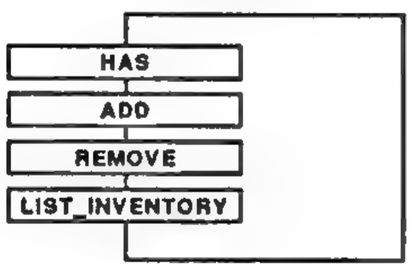
```
procedure GO_RTN (NOUN : in VOCABULARY.NOUNS) is
begin
   if NOUN is a valid direction (N, E, W, S) then
         if the exit is blocked then
               PRINT ("Sorry, you can't go that way");
         else
               Move player  in direction indicated by NOUN.
               Print the appropriate welcoming message.
               List the contents of the new room.
         end if;
   else
         PRINT ("That's really bizarre!!");
   end if;
end GO_RTN;
```


DUNGEON_DOOR



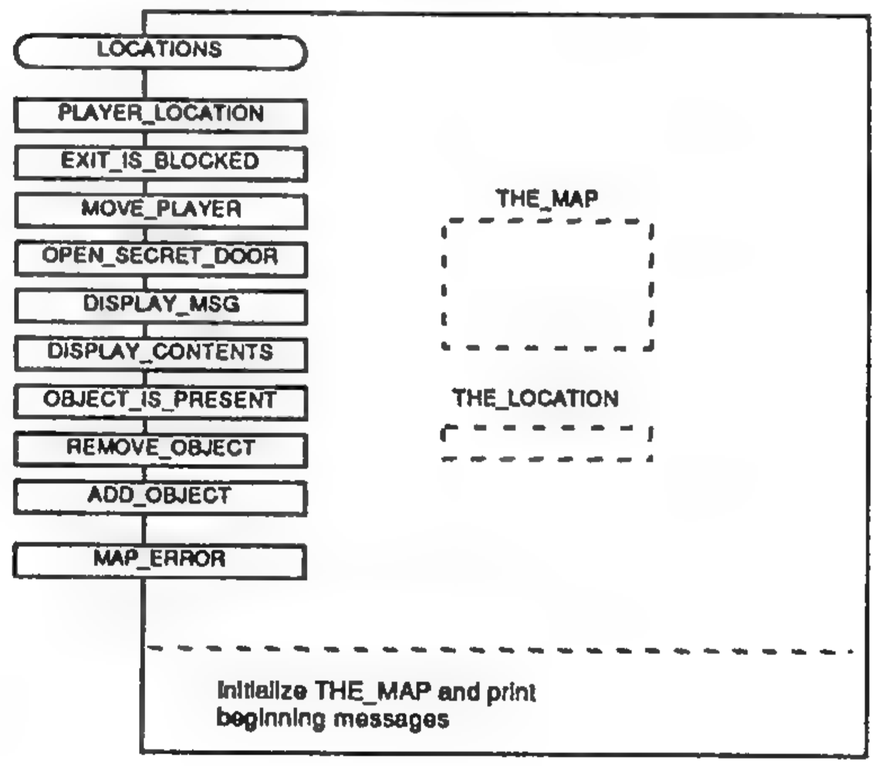

```
package DUNGEON_DOOR is
      procedure OPEN;
      procedure UNLOCK;
      function IS_OPEN return BOOLEAN;
      function IS_LOCKED return BOOLEAN;
end DUNGEON_DOOR;
```


PLAYER

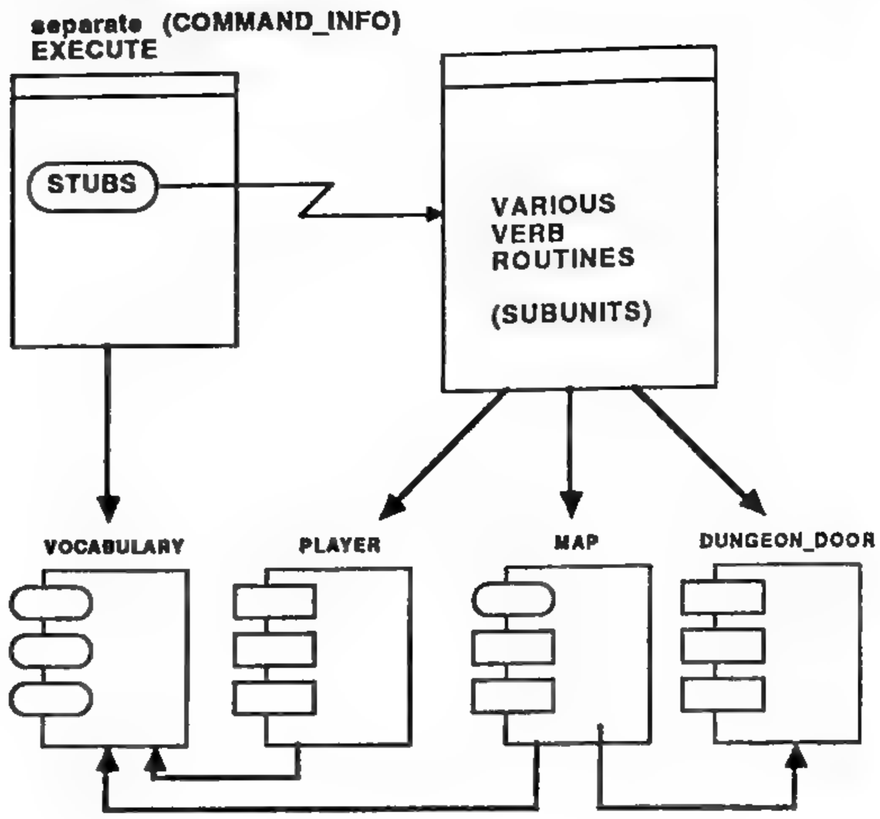



```
with VOCABULARY;
package PLAYER is
   procedure ADD          (OBJECT : In VOCABULARY.THINGS);
   procedure REMOVE  (OBJECT : In VOCABULARY.THINGS);
   function HAS           (OBJECT : VOCABULARY.THINGS)
                                    return BOOLEAN;
   procedure LIST_INVENTORY;
end PLAYER;
```


MAP



```
with VOCABULARY;
package MAP is

    type LOCATIONS is
           (ENTRANCE, CAVE, GOLD_ROOM, GIANT_ROOM,
            CHAMBER, MAZE_1, MAZE_2, MAZE_3, MAZE_4,
            DUNGEON, CELL, BLOCKED);

        -- Note:  all of the following operations are relative to the
        -- current location of the player. That information is kept
        -- in the package body as state information.

    function   PLAYER_LOCATION return LOCATIONS;

    function  EXIT_IS_BLOCKED
              (DIR : VOCABULARY.DIRECTIONS) return BOOLEAN;

    procedure MOVE_PLAYER (DIR : in VOCABULARY.DIRECTIONS);

    procedure  OPEN_SECRET_DOOR;

    procedure  DISPLAY_MSG;

    procedure  DISPLAY_CONTENTS;

    function OBJECT_IS_PRESENT (OBJECT : VOCABULARY.THINGS)
                                    return boolean;

    procedure REMOVE_OBJECT (OBJECT : in VOCABULARY.THINGS);

    procedure ADD_OBJECT          (OBJECT : in  VOCABULARY.THINGS);

    MAP_ERROR : exception;

end MAP;
```


```
with MAP;
separate (COMMAND_INFO.EXECUTE)
procedure GO_RTN (NOUN : in VOCABULARY.NOUNS) is

  begin

     if NOUN in VOCABULARY.DIRECTIONS then  -- N,E,W,S

        if MAP.EXIT_IS_BLOCKED (DIR => NOUN) then

           TEXT_IO.PUT_LINE ("Sorry, you can't go that way");

        else

           MAP.MOVE_PLAYER (DIR => NOUN);
           -- Let the player know where he is
           MAP.DISPLAY_MSG;
           MAP.DISPLAY_CONTENTS;

        end if;

     else

        TEXT_IO.PUT_LINE ("That's really bizarre!!");

     end if;

end GO_RTN;
```



```
package body PLAYER is

     type ITEMS is array (VOCABULARY.THINGS) of BOOLEAN;
     EMPTY_BAG : constant ITEMS := ITEMS'(others=>FALSE);

    THE_BAG : ITEMS := EMPTY_BAG;

    procedure ADD (OBJECT : in VOCABULARY.THINGS) is
     begin
         THE_BAG (OBJECT) := TRUE;
     end ADD;


    procedure REMOVE (OBJECT : in VOCABULARY.THINGS) is
      begin


      end REMOVE;


    function HAS (OBJECT : VOCABULARY.THINGS)
                    return BOOLEAN is
      begin


      end HAS;

      procedure LIST_INVENTORY is separate;

end PLAYER;
```

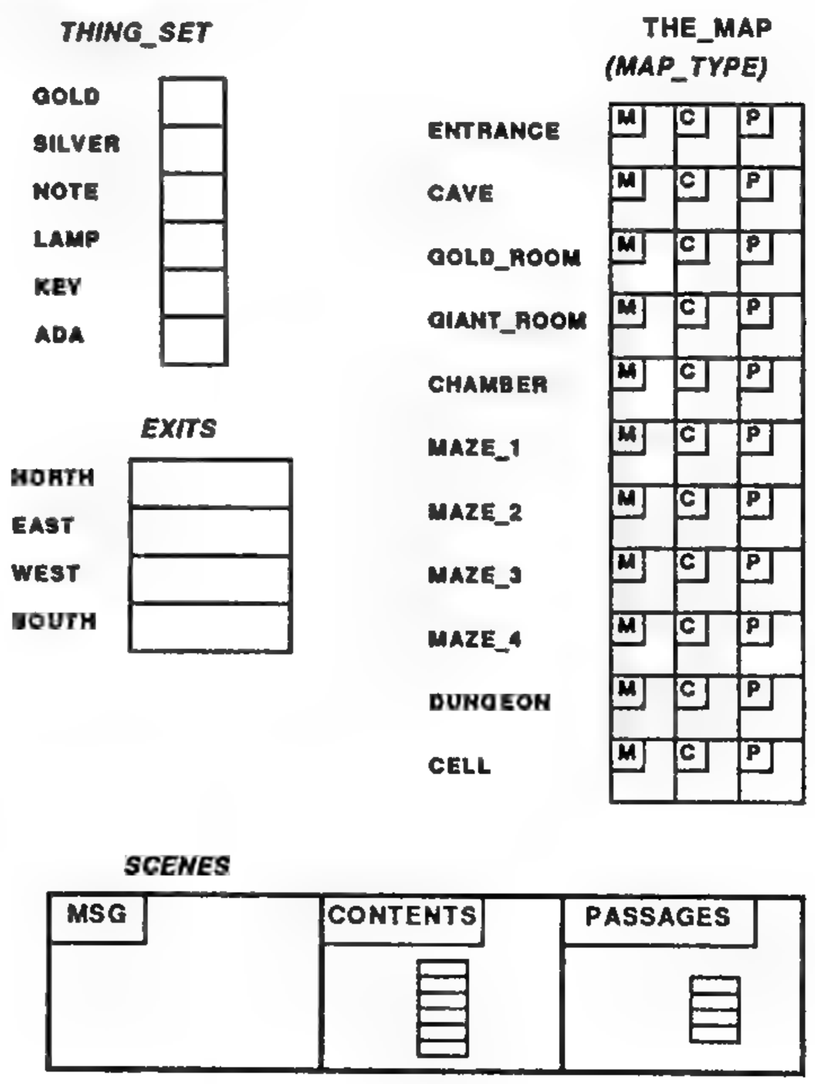
```
with TEXT_IO;
separate (PLAYER)
procedure LIST_INVENTORY is

begin

   if THE_BAG = EMPTY_BAG then

      TEXT_IO.PUT_LINE("You aren't carrying anything");

   else

        TEXT_IO.PUT_LINE("You are carrying the following:");
        for INDEX in VOCABULARY.THINGS
        loop

           if THE_BAG (INDEX) then     -- convert THING
             TEXT_IO.PUT_LINE          -- to STRING
                (VOCABULARY.THINGS'IMAGE (INDEX));
           end if;

        end loop;

   end if;

end LIST_INVENTORY;
```



```
package body DUNGEON_DOOR is

    type STATUS is (OPENED, LOCKED, UNLOCKED);

    THE_DOOR : STATUS := LOCKED;

    procedure OPEN is
    begin
       THE_DOOR := OPENED;
    end;

    procedure UNLOCK is
    begin
       THE_DOOR := UNLOCKED;
    end;

    function IS_OPEN return BOOLEAN is
    begin
       return THE_DOOR = OPENED;
    end;

    function IS_LOCKED return BOOLEAN is
    begin
       return THE_DOOR = LOCKED;
    end;

end DUNGEON_DOOR;
```



```
with DUNGEON_DOOR, TEXT_IO;
package body MAP is

    type THING_SET is array (VOCABULARY.THINGS) of BOOLEAN;

    EMPTY_SET : constant THING_SET := (others => FALSE);

    type EXITS is array (VOCABULARY.DIRECTIONS) of LOCATIONS;

    type SCENES is
       record

          MSG        : STRING (1..60);
          CONTENTS   : THING_SET;
          PASSAGES   : EXITS;

       end record;

    subtype PLACES is LOCATIONS range ENTRANCE .. CELL;

    type MAP_TYPE is array  (PLACES) of SCENES;

-------------------------------------------------------
--------State information follows----------------------
-------------------------------------------------------

   THE_MAP          : MAP_TYPE;

   THE_LOCATION     : PLACES := ENTRANCE;

-------------------------------------------------------
----------Subprogram bodies follow---------------------
-------------------------------------------------------
```

```
function PLAYER_LOCATION return LOCATIONS is
begin


end PLAYER_LOCATION;
-------------------------------------------------------
function EXIT_IS_BLOCKED
        (DIR : VOCABULARY.DIRECTIONS)
             return BOOLEAN is
begin


end EXIT_IS_BLOCKED;
-------------------------------------------------------
procedure MOVE_PLAYER
         (DIR : in VOCABULARY.DIRECTIONS) is
begin


end MOVE_PLAYER;
-------------------------------------------------------
procedure OPEN_SECRET_DOOR is
begin


end OPEN_SECRET_DOOR;
-------------------------------------------------------
procedure DISPLAY_MSG is
begin


end DISPLAY_MSG;
```



```
procedure DISPLAY_CONTENTS is

        procedure PRINT (MSG : STRING) renames TEXT_IO.PUT_LINE;

        use VOCABULARY;     -- to gain visibility of WORDS

begin

     for ITEM in VOCABULARY.THINGS
     loop

        if OBJECT_IS_PRESENT (OBJECT => ITEM) then

          case ITEM is
              when KEY      => PRINT ("There is a key here");
              when NOTE     => PRINT ("There is a note here");
              when LAMP     => PRINT ("There is a lamp here");
              when GOLD     => PRINT ("There is gold here");
              when SILVER   => PRINT ("There is silver here");
              when Ada      => PRINT ("The lovely Ada is here")

          end case;

       end if;

     end loop;              -- for INDEX

      if THE_LOCATION = DUNGEON and
         (not DUNGEON_DOOR.IS_OPEN) then

              PRINT ("A closed door blocks the east exit");

      end if;

  end DISPLAY_CONTENTS;
```



```
function OBJECT_IS_PRESENT (OBJECT : VOCABULARY.THINGS)
                    return BOOLEAN is
begin


end OBJECT_IS_PRESENT;
-------------------------------------------------------
procedure REMOVE_OBJECT (OBJECT : in VOCABULARY.THINGS) is
begin


end REMOVE_OBJECT;
-------------------------------------------------------
procedure ADD_OBJECT (OBJECT : in VOCABULARY.THINGS) is
begin


end ADD_OBJECT;
```


UTILITY ROUTINES

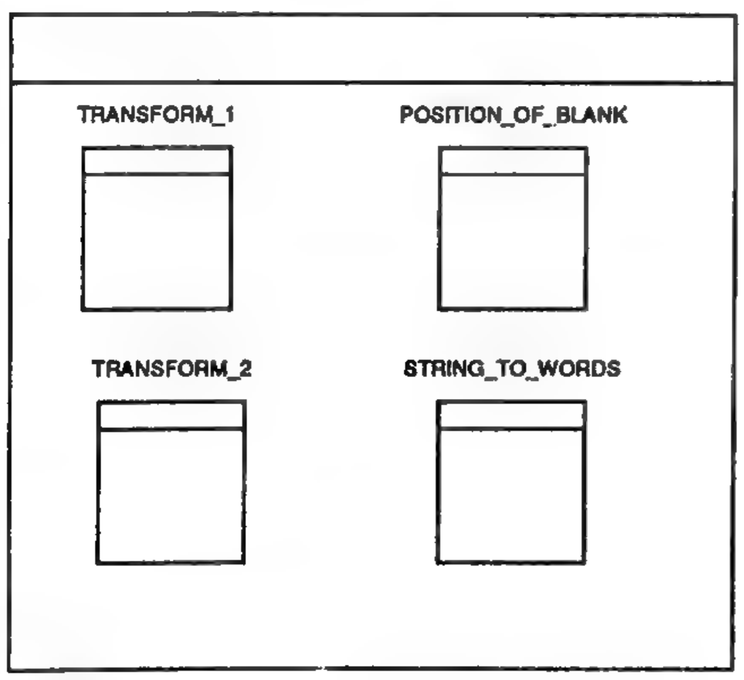
```
-- These three routines are used during the initialization
-- of the data structure.

function PAD (S : STRING) return STRING is

   -- This function converts a smaller string to one which
   -- is constrained to 1 .. 60 (required length of messages).

      RESULT : STRING (1 .. 60) := (1 .. 60 => ' ');

begin
       RESULT (1 .. S'LAST) := S;
      return RESULT;
end PAD;


function INIT (OBJ : VOCABULARY.THINGS) return THING_SET is
       COLLECTION : THING_SET := EMPTY_SET;
begin
       COLLECTION (OBJ) := TRUE;
       return COLLECTION;
 end INIT;


function INIT (OBJ1,OBJ2 : VOCABULARY.THINGS) return THING_SET is
       COLLECTION : THING_SET := EMPTY_SET;
begin
      COLLECTION (OBJ1) := TRUE;
      COLLECTION (OBJ2) := TRUE;
       return COLLECTION;
end INIT;
```

```
use VOCABULARY;  -- Direct visibility of WORDS

begin

   -- This is the optional sequence of statements which
   -- is executed when the package body is
   -- elaborated.  It is used to set up THE_MAP.


    THE_MAP (ENTRANCE) :=
         (PAD ("You are at the entrance of a cave"),
          INIT (NOTE,LAMP),
          (BLOCKED, BLOCKED, BLOCKED, CAVE)  );

    THE_MAP (CAVE) :=
         (PAD ("You are in a large cave"),
          EMPTY_SET,
          (ENTRANCE, GIANT_ROOM,
           BLOCKED, GOLD_ROOM)  );

    THE_MAP (GOLD_ROOM) :=
         (PAD ("You have entered the gold room"),
          INIT(GOLD),
          (BLOCKED, BLOCKED, CAVE, BLOCKED)  );

    THE_MAP (GIANT_ROOM) :=
         (PAD (                                    ),
          EMPTY_SET,
          (                                        ) );
```



```
    THE_MAP (CHAMBER) :=


    THE_MAP (MAZE_1) :=


    THE_MAP (MAZE_2) :=


    THE_MAP (MAZE_3) :=


    THE_MAP (MAZE_4) :=


    THE_MAP (DUNGEON) :=


    THE_MAP (CELL) :=


    DISPLAY_MSG;              -- initial information
    DISPLAY_CONTENTS;         -- for the player

end MAP;
```


GET

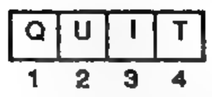


Read an input string.
If there is no blank,
    attempt to generate a one word command
else attempt to generate a two-word command.
If an error occurs,
    generate an appropriate message and
    repeat the entire process.

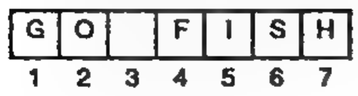

```
with TEXT_IO;
separate  (COMMAND_INFO)
procedure GET (CMD : out COMMAND) is

    STR       : STRING (1..30);     -- actual user input
    COUNT     : NATURAL;            -- actual # of characters in STR
    SPOT      : NATURAL;            -- position of blank (if any)

    BAD_COMMAND : exception;

    function POSITION_OF_BLANK (WITHIN : STRING)
                  return NATURAL is separate;

    function STRING_TO_WORDS (S : STRING)
                  return VOCABULARY.WORDS is separate;

    function TRANSFORM_1 (S : STRING)
                  return COMMAND is separate;

    function TRANSFORM_2 (V, N : STRING)
                  return COMMAND is separate;
begin

        TEXT_IO.GET_LINE (STR, COUNT);

        SPOT := POSITION_OF_BLANK (STR (1..COUNT));

        if SPOT = 0 then
            CMD := TRANSFORM_1 (STR (1..COUNT));
        else
            CMD := TRANSFORM_2 (STR (1..SPOT-1),
                                STR (SPOT + 1..COUNT));
        end if;

exception

         when BAD_COMMAND => GET(CMD);       -- recursive

end GET;
```

-- VALID

-- VALID

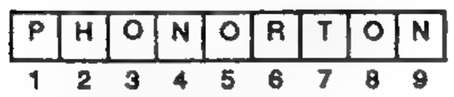


-- VALID

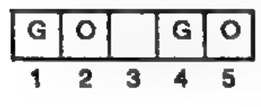


-- ILLEGAL

-- INVALID

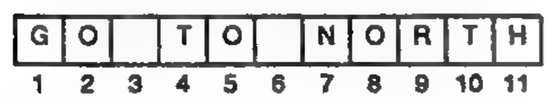


-- ILLEGAL



-- ILLEGAL

-- ILLEGAL



-- ILLEGAL


```
separate (COMMAND_INFO.GET)
function POSITION_OF_BLANK (WITHIN : STRING)
            return NATURAL is

        -- This function returns the ordinal position
        -- of the first blank in the string and, if
        -- no blank is found, returns zero.

begin

    for INDEX in WITHIN'RANGE
    loop

        if WITHIN (INDEX) = ' ' then
            return INDEX;
        end if;

    end loop;

    return 0;

end POSITION_OF_BLANK;
```



```
separate (COMMAND_INFO.GET)
function TRANSFORM_1 (S : STRING) return COMMAND is

  use VOCABULARY;
  THE_WORD : WORDS;     -- holds converted noun or verb

begin

  THE_WORD := STRING_TO_WORDS (S);
              -- if no exception, THE_WORD is legal

  case THE_WORD is

     when QUIT | INVENTORY =>
              return (THE_WORD, NORTH)  -- NORTH is arbitrary

     when NORTH .. SOUTH =>
              return (GO,THE_WORD);

     when others =>
              raise BAD_COMMAND;

  end case;

exception

  when BAD_COMMAND =>
      TEXT_IO.PUT_LINE ("I don't understand that command");
      raise;

end TRANSFORM_1;
```



```
separate (COMMAND_INFO.GET)
function TRANSFORM_2 (V, N : STRING) return COMMAND is

    -- This function simply returns an  agreggate value.
    -- an exception will result if there is no match for
    -- either the VERB or the NOUN.

begin
    return (STRING_TO_WORDS (V),
            STRING_TO_WORDS (N) );
exception
  -- First, check for out-of-order conditions

     when CONSTRAINT_ERROR =>

        TEXT_IO.PUT_LINE ("I don't understand");
        raise BAD_COMMAND;

  -- process 'SAY' command while still a string
    when BAD_COMMAND =>

      if V = "SAY" then

        if N = "abracADAbra" then
            return (SAY, MAGIC_WORD);
        else
            TEXT_IO.PUT("OK . . .");
            TEXT_IO.PUT_LINE (N);
        end if;

      else
          TEXT_IO.PUT_LINE("I don't understand");
      end if;
      raise;

end TRANSFORM_2;
```

```
separate  (COMMAND_INFO.GET)
function STRING_TO_WORDS (S : STRING)
                              return VOCABULARY.WORDS is

   -- This function uses the 'value' attribute to convert from
   -- string to type WORDS. The attribute, by definition, raises
   -- a constraint_error if no conversion is possible.

begin

   return VOCABULARY.WORDS'VALUE (S);

exception

   when CONSTRAINT_ERROR =>
      raise BAD_COMMAND;

end STRING_TO_WORDS;
```


USING A SYNONYM TABLE

NORTH	NORTH
N	NORTH
EAST	EAST
E	EAST
WEST	WEST
W	WEST
SOUTH	SOUTH
S	SOUTH
GOLD	GOLD
SILVER	SILVER
NOTE	NOTE
LAMP	LAMP
LANTERN	LAMP
KEY	KEY
ADA	ADA
DOOR	DOOR

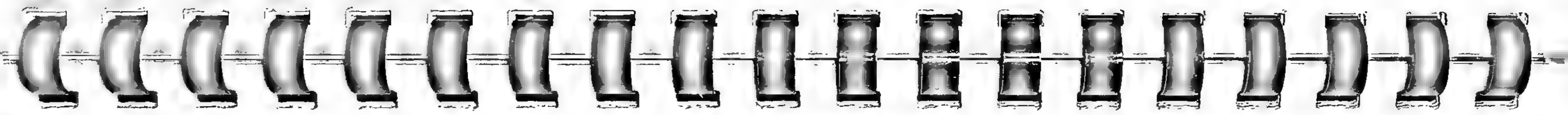
GO	GO
MOVE	GO
TAKE	TAKE
GRAB	TAKE
GET	TAKE
LIGHT	LIGHT
DROP	DROP
THROW	DROP
PUT	DROP
DISCARD	DROP
READ	READ
SAY	SAY
OPEN	OPEN
UNLOCK	UNLOCK
QUIT	QUIT
Q	QUIT
INVENTORY	INVENTORY
INVENT	INVENTORY
INV	INVENTORY

ALLOWING SYNONYMS

```
separate  (COMMAND_INFO.GET)
function STRING_TO_WORDS (S : STRING) return WORDS is

   type SYNONYM is
      (NORTH, N, EAST, E, WEST, W, SOUTH, S, GOLD,
       SILVER, NOTE, LAMP, LANTERN, KEY, ADA, DOOR,
       GO, MOVE, TAKE, GRAB, GET, LIGHT, DROP,
       THROW, PUT, DISCARD, READ, SAY, OPEN,
       UNLOCK, QUIT, Q, INVENTORY, INVENT, INV);

   use VOCABULARY;

   TABLE : array (SYNONYM) of VOCABULARY.WORDS:=
      (NORTH, NORTH, EAST, EAST, WEST, WEST
       SOUTH, SOUTH, GOLD, SILVER, NOTE, LAMP,
       LAMP, KEY, ADA, DOOR, GO, GO, TAKE, TAKE,
       TAKE, LIGHT, DROP, DROP, DROP, DROP, READ,
       SAY, OPEN, UNLOCK, QUIT, QUIT, INVENTORY,
       INVENTORY, INVENTORY);

begin

         return TABLE (SYNONYM'VALUE (S) );

exception

         when CONSTRAINT_ERROR =>
               raise BAD_COMMAND;

end  STRING_TO_WORDS;
```



```
separate (COMMAND_INFO)
function USER_QUITS (C : COMMAND) return BOOLEAN is

   use VOCABULARY;

   -- You might want to interact with the user to determine
   -- his/her actual wishes

begin

   return C.VERB = QUIT;

end USER_QUITS;

-----------------------------------------------

separate (COMMAND_INFO)
function USER_WINS return BOOLEAN is

begin

   -- an algorithm to assess winning criteria
   -- would go here

end USER_WINS;
```



Exercises 1

1. Scalar Types

   a. Declare an Integer type to represent lines on a CRT.

   Type CRT_Line is range 1 .. 24;

   b. Declare an object of the above type initialized to 24 lines.

   My_Line : CRT_Line := 24;

   c. Declare a floating-point type with 9 digits of precision.

   Type My_FLOAT is digits 9;

   d. Declare a fixed-point type which will represent voltages between 10.0 and 2000.0 volts with a granularity of 1/4 volt.

   Type Volts is delta 0.25 rang 10.0 .. 2000.0;

   e. Declare an enumeration type whose literals are the two-letter postal codes of the Confederate States of America.

   Type ~~COX~~ _CODE Is (AL, FL, GA, LA, VA, ~~KY~~, ~~MO~~, TX
   SC, NC, AR, TN, MS)

   f. Declare a subtype of the above type containing only those Confederate states which are completely land-locked.

   Subtype Land_lock is CON_CODE range AR .. TN;

   g. Declare a character type (enumeration) for ranks of playing cards. Disregard the joker.

   (52)

   Type Card is ('2', 3, '4', '5', '6', '7', '8', '9', '10', J, Q, K, A)

2. Composite Types

a. Declare an array type for casualties incurred by each state of the Confederate States of America.

b. Declare an object of the type with all values initially 0.

c. Write an assignment statement indicating that Georgia had 13,597 casualties.

d. Declare a string constant which contains your name in the form:
<first name><space><initial><.><space><last name>

e. Declare a string variable (not a constant) which contains as an initial value your name in the form:
<last name><,><space><first name><space><initial><.>

The catch: With the exception of <,> you may use only catenation (&) and slices from the string constant declared in d. above.

f. Declare a record type for complex numbers.

INPUT/OUTPUT PRIMER

1. Any program unit (procedure, package etc.) which does input/output operations should have the following context specification:

```
with TEXT_IO;
procedure <identifier> is
```

This allows the application programmer the capability of inputting and outputting values of the predefined types STRING and CHARACTER.

2. To input and output the predefined type INTEGER, the following declaration must appear within the declarative part of the procedure or package which will perform the operation:

```
package INT_IO is new TEXT_IO.INTEGER_IO (INTEGER);
```

3. To input and output values of the enumerated data type

```
type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);
```

the following declaration must appear within the declarative part of the procedure or package which will perform the operation:

```
package DAYS_IO is new TEXT_IO.ENUMERATION_IO (DAYS);
```

5. Given the above, the following are all valid statements:

```
TEXT_IO.PUT ("This is a string literal");
TEXT_IO.PUT_LINE ("Only strings can use ""PUT_LINE"" ");
INT_IO.PUT(17);
INT_IO.PUT(17,5);        -- right justified in a field of length 5
DAYS_IO.PUT(WED);
TEXT_IO.NEW_LINE;   -- generates CR,LF for any data type
```

1. Given the following declarations:

```
type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);

type LIST is array (DAYS) of NATURAL;

MY_LIST : LIST := (2,4,6,8,10,12,14);
```

write and execute an Ada procedure which will

a. Output the value of the following attributes for type DAYS:

FIRST	**LAST**	**PRED (MON)**
SUCC (MON)	**VAL (2)**	**VALUE ("WED")**
POS (FRI)	**IMAGE (SAT)**	

NOTE: The first six are of type DAYS, the seventh of type universal integer and the last is of type STRING

b. Output the value of the following attributes for type LIST:

FIRST	**LAST**	**LENGTH**

c. Output the values of MY_LIST

2. Write a program which will print out all 3-digit numbers xyz (000-999) which have the property that xyz = x**n + y**n + z**n. The user of the program should be able to enter a value for n, receive a report and continue entering other values for n. The program should accept values of n as large as 10. the program should terminate when the user enters a value of zero.

3. Write a boolean function which accepts a string and determines if the string is a palindrome (reads the same forwards and backwards). The strings should be one word long and palindromes, in our case, are case sensitive. That is, "ADA" and "radar" are palindromes while "Ada" and "PHONORTON" are not. Compile the function and then write a driver program which calls the function.

4. Given the following types:

```
type COLOR   is (RED, BLUE, GREEN, MAGENTA, PURPLE);
type LIGHT   is (RED, GREEN, AMBER);
```

write a program which contains a function which will convert from type COLOR to type LIGHT. That is, if the argument (of type COLOR) to the function represents an enumeration value whose value also appears in type LIGHT, then the conversion will be made successfully. If there is no corresponding value, a constraint error should be raised.

5. Write a program which will ring the bell 5 times.

CALENDAR

1. Write the package body to implement the following specification of the package CALENDAR_INFO. Add any utility routines to the body which you think might be helpful. The output should be in the form shown below.

2. Write a driver program which has only the following two statements:

```
PRINT_MONTH  (1988,  FEB,  MON);
PRINT_MONTH  (1987,DEC,  TUE);
```

3. Modify the program to allow user selection of month, day and year.

```
package CALENDAR_INFO is

   type DAYS is (SUN, MON, TUE, WED, THU, FRI, SAT);

   type MONTHS is ( JAN, FEB, MAR, APR, MAY, JUN,
                    JUL, AUG, SEP, OCT, NOV, DEC);

   subtype YEARS is NATURAL range 1901 .. 2099;

   procedure  PRINT_MONTH ( YEAR    : in YEARS;
                            MONTH : in MONTHS;
                            START  : in DAYS );


end CALENDAR_INFO;
```

```
FEB                                    1986

 S    M    T    W    T    F    S
                                      1
 2    3    4    5    6    7    8
 9   10   11   12   13   14   15
16   17   18   19   20   21   22
23   24   25   26   27   28
```

ROMAN NUMERAL

1. Write the body to implement the following package specification. The ROMAN NUMERALS are to follow the ancient form (9 = VIIII not IX).

2. Write a driver program which will exercise all of the operations in the package.

```
package ROMAN is

    type DIGIT is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

    type DIGIT_STRING is array (POSITIVE range <>) of DIGIT;

        -- By definition DIGIT_STRINGs contain only DIGITS.
        -- "II", "IVI", XVXIX" ( but not "XVIAV") are legal DIGIT_STRINGS.

    type NUMERAL is private;

    type VALID_NUMBER is range 1 .. 4999;

    ILLEGAL_ROMAN_NUMERAL : exception;

        -- raised when illegal characters, converted number greater
        -- than 4999, empty input, invalid ordering of DIGITs or too many
        -- of a given DIGIT.

    procedure GET_VALID (RN : out NUMERAL);

        -- Interacts with user in order to input a valid roman numeral.
        -- ILLEGAL_ROMAN_NUMERAL can be raised.

    procedure PUT (RN : in NUMERAL);

        -- Outputs a ROMAN NUMERAL.  No carriage return.

    function CREATE (S : DIGIT_STRING) return NUMERAL;

        -- ILLEGAL_ROMAN_NUMERAL will be raised if DIGITs are out
        -- of order or if there are too many of a given DIGIT.
```

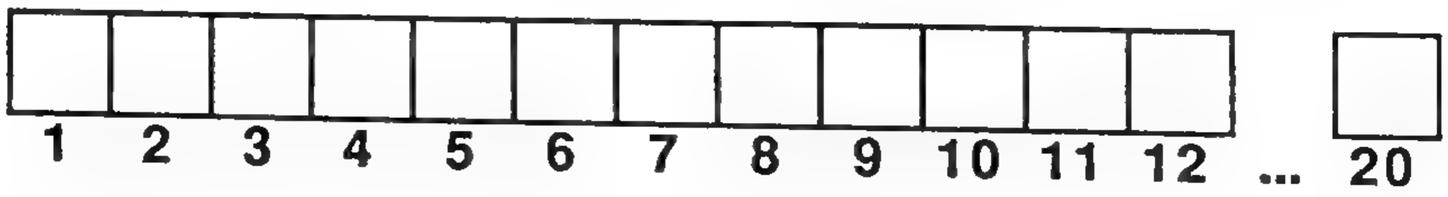
ROMAN NUMERAL

```
function "+" ( LEFT, RIGHT : NUMERAL) return NUMERAL;
        -- ILLEGAL_ROMAN_NUMERAL will be raised if sum exceeds 4999.

function "<" (LEFT, RIGHT : NUMERAL) return BOOLEAN;

    function CONVERT (RN : NUMERAL) return VALID_NUMBER;
            -- RN of "VII" returns 7
            -- RN of "MMMMDCCCCLXXXXVIIII" returns 4999
    function CONVERT (VN : VALID_NUMBER) return NUMERAL;
            -- VN of 7 returns "VII"
            -- VN of 4999 returns "MMMMDCCCCLXXXXVIIII"

private
    type NUMERAL is
            record
                SIZE : NATURAL;   -- Number of actual DIGITs
                LIST : DIGIT_STRING (1 .. 20);
            end record;

end ROMAN;
```

SIZE

for a VALID_NUMBER of 7, LIST (1 .. 3) = "VII" and SIZE = 3.
for a VALID_NUMBER of 2016, LIST (1 .. 5) = "MMXVI" and SIZE = 5.

LIST



CHANGE MAKER

1. Given the following generic package specification for CHANGE_INFO, write the package body.

2. Write a program which will use the generic package to provide change-making capability for United States currency using the following type:

```
type DENOM is ( PENNY, NICKEL, DIME, QUARTER, HALF,
                ONE, FIVE, TEN, TWENTY, FIFTY );
```

3. Modify the program so that it will provide change-making capability for currency for some other country. If you do not know the currency of another country, make up something.

NOTES:

The user should be allowed to enter as many pairs of values as he/she wishes

Values should not exceed 1000.00.

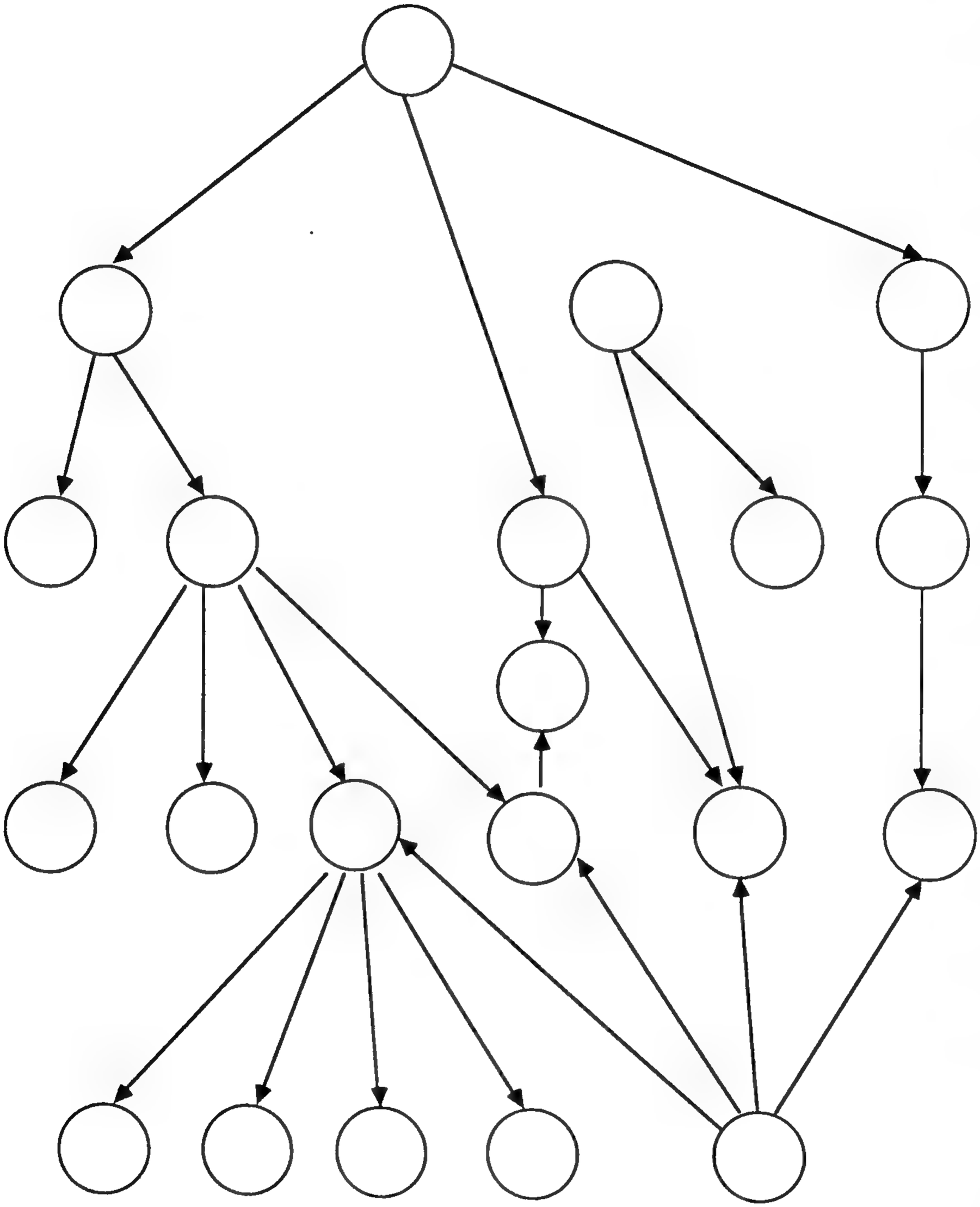
Values should be entered with exactly 2 decimal places. (You may assume that the input has the correct number of decimal places. You need not validate this.)

If the amount offered is less than the amount charged, the user should be informed and allowed to enter another pair of amounts.

CHANGE MAKER

```
generic

        type CURRENCY_NAMES is (<>);
        type CURRENCY_LIST is array (CURRENCY_NAMES) of NATURAL;
        CURRENCY_VALUES   :   in CURRENCY_LIST;

                -- CURRENCY_NAMES must be ordered 'low-to-high'
                -- CURRENCY_VALUES represent canonical values for
                -- each denomination (TWENTY = 2000, etc.)

package CHANGE_INFO is

        subtype CANONICAL_UNITS is NATURAL range 0 .. 100_000;
        type MONEY_TYPE is digits 5 range 0.0 .. 1_000.0;

        procedure GET_INPUT ( PRICE : out MONEY_TYPE;
                              PAID  : out MONEY_TYPE );

                -- Interactively gets input from the user.  The user will be allowed
                -- to reenter a data value in case of error.  PAID must not be less than PRICE.

        function CHANGE_DUE ( PRICE : MONEY_TYPE;
                              PAID  : MONEY_TYPE )
                                    return CANONICAL_UNITS;


        function MAKE_CHANGE (UNITS : CANONICAL_UNITS)
                                    return CURRENCY_LIST;

                -- Takes a value of CANONICAL_UNITS (perhaps pfennigs) and
                -- creates an array value which contains the appropriate number
                -- of each denomination to be issued in change.


        procedure PRINT_CURRENCY (MONEY : in CURRENCY_LIST);

        function USER_WANTS_TO_STOP return BOOLEAN;

                -- Interacts with the user to determine if any more pairs of values
                -- will be forthcoming.

end CHANGE_INFO;
```

	CAT	DEP	RULE
1. procedure PLAY_AdaVENTURE			
2. package COMMAND_INFO	LIB	4	1
3. package body COMMAND_INFO	SEC	2	2
4. package VOCABULARY			
5. procedure GET			
6. procedure EXECUTE	SEC	3/21	3/1
7. function USER_QUITS			
8. function USER_WINS			
9. procedure GO_RTN			
10. package PLAYER			
11. package body PLAYER			
12. package DUNGEON_DOOR			
13. package body DUNGEON_DOOR			
14. package MAP			
15. package body MAP			
16. procedure LIST_INVENTORY			
17. function POSITION_OF_BLANK			
18. function STRING_TO_WORDS			
19. function TRANSFORM_1			
20. function TRANSFORM_2			
21. package TEXT_IO			



Exercise 1

A simple random number generator yielding a random number (RN) between 0.0 and 1.0 is:

SEED ← (SEED * 824) MOD 10657

RN ← SEED / 10657 (This is 'real' division)

1. Using the above algorithm, Implement a random number capability which will go into your library and be availble for use. The random number generator is to get the initial SEED value from the user of the function. The initial seed must be a five-digit odd integer.

2. Test the random number generator by writing a program to generate 50 floating point numbers between 0.0 and 1.0;

3. Test the random number generator by writing a program which will generate 1000 integers between 0 and 9 and will print out a report of their frequencies.

4. Write a program to generate random values from the following type:

type Days is (SUN, MON, TUE, WED, THU, FRI, SAT);

Test the program as in 3 above.

5. Write a generic random capability which will work for any discrete type.

Exercise 2

Using object-oriented design, design, implement and test a generic **queue** package. The element type and the maximum number of elements should be passed as generic formal parameters.

1. **OBJECT**: Queue

2. **CONSTRUCTORS** **EXCEPTIONS (If any)**

3. **SELECTORS** **EXCEPTIONS (If any)**

4. **REQUIRED FROM CLIENT:**

   a. **Element type**
   b. **Maximum size of queue**

5. **OUTSIDE VIEW (Package spec)**

Exercise 3

1. Write a program containing four tasks. PRODUCER_1 sends strings to CONSUMER_1 and PRODUCER_2 sends strings to CONSUMER_2. The two consumer tasks will contain the entry declarations and the producer tasks will contain the calls.

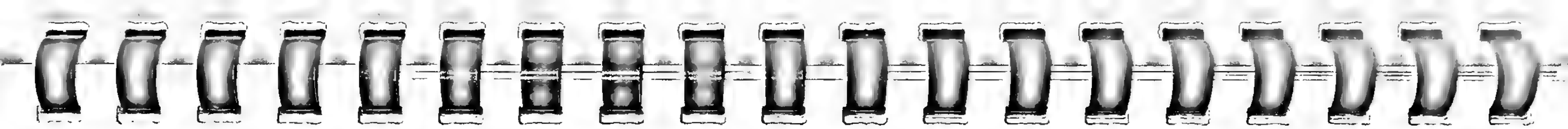
   a. The two producer tasks should send their strings at an interval between one and two seconds (determined by a random number). Each producer task should send five messages. Each message should contain (at least) the name of the producer task.

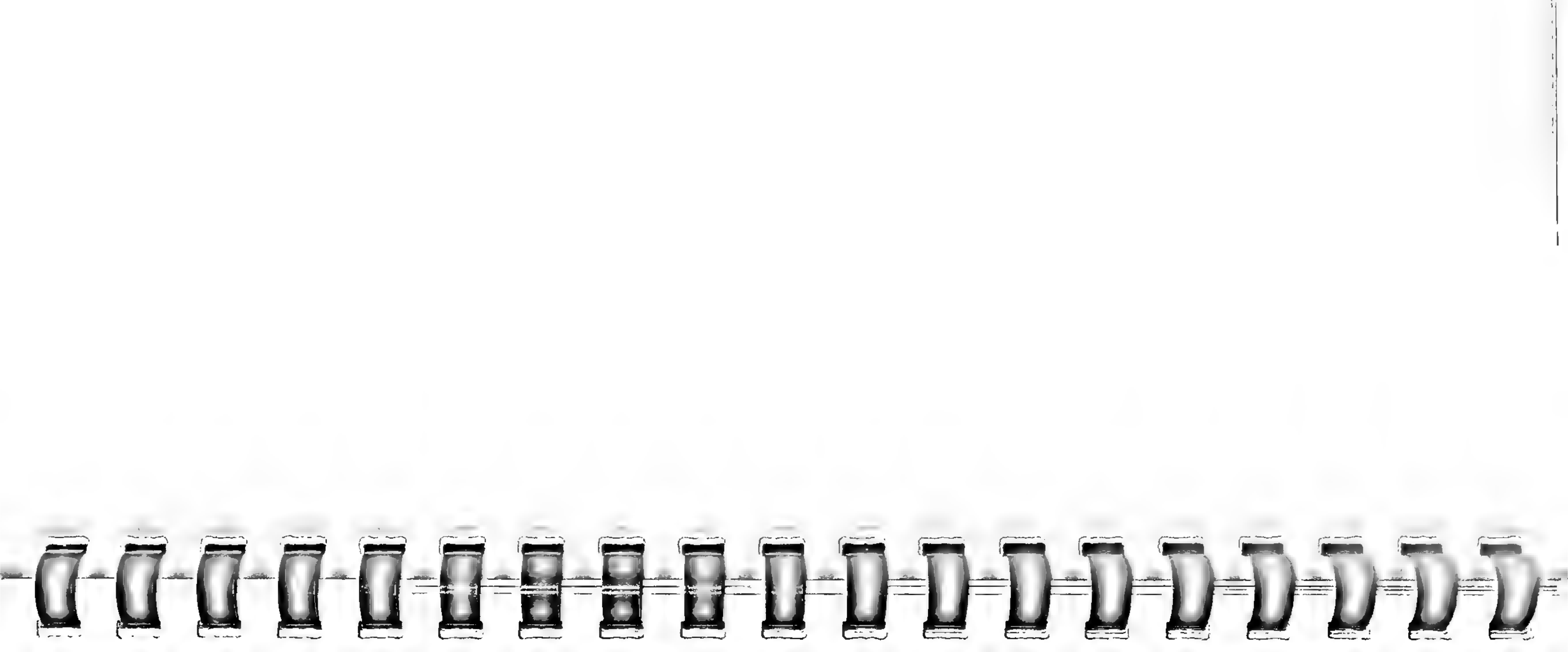
   b. The two consumer tasks should print each message as soon as it is received. The consumer task should append the name of the consumer task to the message prior to printing. Sample output might look like this:

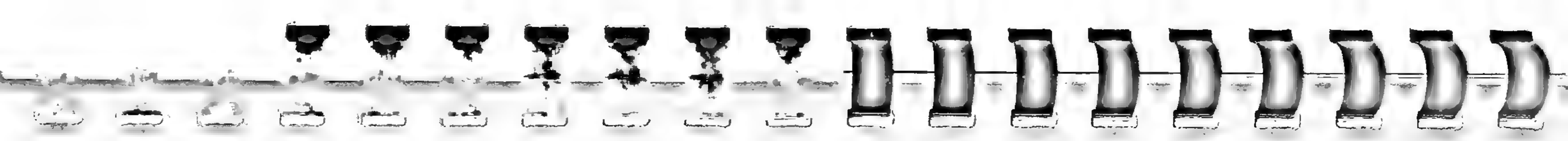
   MSG 5 FROM PRODUCER 1///CONSUMER 1

2. Modify the above system so that the producer tasks contain the entry declarations and the consumer tasks contain the calls

3. Modify the above system to insert a buffer task between the consumers and producers. Thus, the two producers will send messages to the buffer task, not knowing which consumer will pick them up. The two consumer tasks will pick up messages from the buffer task without knowing which producer sent them. The buffer task should buffer up at most 4 messages. In this case, the buffer tasks will contain the entry declarations and the producer and consumer tasks will contain the calls.







1. Scalar Types

a. Declare an integer type to represent lines on a CRT.

```
type CRT_LINES is range 0 .. 66;
```

b. Declare an object of the above type initialized to 24 lines.

```
VT_100_MAX : CRT_LINES := 24;
```

c. Declare a floating-point type with 9 digits of precision.

```
type MY_FLOAT is digits 9;
```

d. Declare a fixed-point type which will represent voltages between 10.0 and 2000.0 volts with a granularity of 1/4 volt.

```
type VOLTS is delta 0.25 range 10.0 .. 2000.0;
```

e. Declare an enumeration type whose literals are the two-letter postal codes of the Confederate States of America.

```
type CSA is (LA, AL, NC, SC, TN, AR, VA, TX, FL, MS, GA);
```

f. Declare a subtype of the above type containing only those Confederate states which are completely land-locked.

```
subtype LAND_LOCKED is CSA range TN .. AR;
```

g. Declare a character type (enumeration) for ranks of playing cards. Disregard the joker.

```
type RANKS is ('2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K', 'A');
```


2. Composite Types

a. Declare an array type for casualties incurred by each state of the Confederate States of America.

```
type CASUALTIES is array (CSA) of natural;
```

b. Declare an object of the type with all values initially 0.

```
FATAL : CASUALTIES := (CASUALTIES'RANGE => 0);
```

c. Write an assignment statement indicating that Georgia had 13,597 casualties.

```
FATAL (GA) := 13_597;
```

d. Declare a string constant which contains your name in the form:
<first name><space><initial><.><space><last name>

```
MY_NAME : constant STRING := "Richard E. Botz";
```

e. Declare a string variable (not a constant) which contains as an initial value your name in the form:
<last name><,><space><first name><space><initial><.>

```
THE_NAME : STRING(1..16) := MY_NAME (12 .. 15) & ',' &
                            MY_NAME (8 .. 8) & MY_NAME (1 .. 10);
```

The catch: With the exception of <,> you may use only catenation (&) and slices from the string constant declared in d. above.

f. Declare a record type for complex numbers.

```
type COMPLEX is
   record
      REAL_PART : FLOAT := 0.0;
      IMAG_PART : FLOAT := 0.0;
   end record;
```


THREE-DIGIT NUMBER PROBLEM (EXERCISE PG 5)

```
with TEXT_IO;
procedure THREE_DIGIT is

      THE_NUMBER     : NATURAL range 0 .. 999;
      N              : NATURAL range 0 .. 10;      -- The power

      package INT_IO is new TEXT_IO.INTEGER_IO (NATURAL);

begin
   loop
      begin
            TEXT_IO.PUT_LINE ("Enter Power (0 to quit):");
            INT_IO.GET(N);
            exit when N = 0;

            TEXT_IO.PUT ("For N = ");
            INT_IO.PUT (N,2);
            TEXT_IO.PUT_LINE (" the values are:");

            for X in 0 .. 9 loop
                  for Y in 0 .. 9 loop

                        for Z in 0 .. 9 loop

                              THE_NUMBER := X*100 + Y*10 + Z;
                              if THE_NUMBER = X**N + Y**N + Z**N then
                                    INT_IO.PUT (THE_NUMBER);
                                    TEXT_IO.NEW_LINE;
                              end if;
                        end loop;   -- for Z

                  end loop;  -- for Y

            end loop; -- for X
      exception
         when TEXT_IO.DATA_ERROR | CONSTRAINT_ERROR =>
               TEXT_IO.PUT_LINE ("Invalid power.  Restart process.");

      end; -- block

   end loop;
end THREE_DIGIT;
```


PALINDROME PROBLEM (EXERCISE PG 5)

```
function IS_PALINDROME (STR : STRING) return BOOLEAN is

      MIRROR_IMAGE : STRING (STR'FIRST .. STR'LAST);

begin
      for INDEX in 1 .. STR'LAST
      loop
         MIRROR_IMAGE (INDEX) := STR ( (STR'LAST - INDEX) + 1);
      end loop;

      return STR = MIRROR_IMAGE;

end IS_PALINDROME;
```

```
with TEXT_IO, IS_PALINDROME;
procedure PALINDROME_CHECK is

      S : STRING (1 .. 30);
      COUNT : NATURAL;

begin
      loop
            TEXT_IO.PUT_LINE ("Enter a word (<CR> to quit)");
            TEXT_IO.GET_LINE (S, COUNT);
            exit when COUNT = 0;

            TEXT_IO.PUT (S (1 .. COUNT));

            if IS_PALINDROME (S (1 .. COUNT)) then
               TEXT_IO.PUT_LINE (" is a palindrome");
            else
               TEXT_IO.PUT_LINE (" is not a palindrome");
            end if;

      end loop;

end PALINDROME_CHECK;
```

CONVERSION PROBLEM (EXERCISE PG 5)

```
with TEXT_IO;
procedure CONVERSION is

    type COLOR is (RED, BLUE, GREEN, MAGENTA, PURPLE);
    type LIGHT  is (RED, GREEN, AMBER);

    function CONVERT (C : COLOR) return LIGHT is
    begin
        return LIGHT'VALUE (COLOR'IMAGE (C));
    end CONVERT;
begin
    for HUE in COLOR            -- COLOR'FIRST .. COLOR'LAST
    loop

        begin    -- block statement encapsulates exception handler
            TEXT_IO.PUT (LIGHT'IMAGE (CONVERT (HUE)));
            TEXT_IO.PUT_LINE (" is in both types.");
        exception
            when CONSTRAINT_ERROR =>
                TEXT_IO.PUT (COLOR'IMAGE (HUE));
                TEXT_IO.PUT_LINE (" is in type COLOR only.");
        end;
    end loop;
end CONVERSION;
```


TEXT PROBLEM 1 (STUDENT NOTES PG 189)

```
with BOUNDED_LENGTH_STRING, TEXT_IO;
use BOUNDED_LENGTH_STRING;
procedure SUBSTITUTE is

    THE_TEXT  : TEXT;
    SPOT      : INDEX;
begin

    GET (THE_TEXT);

    SPOT := POS (PATTERN => "FRAMUS", SOURCE => THE_TEXT);

    if SPOT /= 0 then

        DELETE ( ORIGINAL => THE_TEXT,
                 START    => SPOT,
                 COUNT    => 6 );

        INSERT ( SOURCE   => "PHONORTON",
                 ORIGINAL => THE_TEXT,
                 START    => SPOT );
    end if;

    PUT (THE_TEXT);

exception
    when SIZE_ERROR =>

        TEXT_IO.PUT_LINE ("TEXT too large");
end SUBSTITUTE;
```


TEXT PROBLEM 2 (STUDENT NOTES PG 190)

```
with BOUNDED_LENGTH_STRING, TEXT_IO;
use BOUNDED_LENGTH_STRING;
procedure ONE_PER_LINE is

    THE_TEXT  : TEXT;
    LEFT      : INDEX := 1;
    RIGHT     : INDEX;
begin

    GET (THE_TEXT);

    if LENGTH (THE_TEXT) /= 0 then

      loop
        RIGHT := POS (" ", THE_TEXT, START => LEFT);

        exit when RIGHT = 0;

        PUT_LINE ( COPY (SOURCE => THE_TEXT,
                         START  => LEFT,
                         COUNT  => SIZE (RIGHT - LEFT)));
        LEFT := RIGHT + 1;

      end loop;

      -- Output the final word

      PUT_LINE (COPY ( SOURCE => THE_TEXT,
                       START  => LEFT,
                       COUNT  => LENGTH (THE_TEXT) -
                                 SIZE (LEFT) + 1));
    end if;
exception
    when others =>

        TEXT_IO.PUT_LINE ("Unknown error");
end ONE_PER_LINE;
```

CALENDAR PROBLEM (EXERCISE PG 6)

```
package body CALENDAR_INFO is

    subtype DAY_RANGE is NATURAL range 1 .. 31;

    function LAST_DAY (Y : YEARS; M : MONTHS) return DAY_RANGE is

    begin

        case M is

            when SEP | APR | JUN | NOV => return 30;

            when FEB =>

                if Y mod 4 = 0 then
                    return 29;
                else
                    return 28;
                end if;

            when others => return 31;

        end case;

    end LAST_DAY;


    procedure PRINT_MONTH ( YEAR   : in YEARS;
                            MONTH  : in MONTHS;
                            START  : in DAYS ) is separate;

end CALENDAR_INFO;
```


CALENDAR PROBLEM (EXERCISE PG 6)

```
with TEXT_IO;
separate (CALENDAR_INFO)
procedure PRINT_MONTH (YEAR : in YEARS; MONTH : in MONTHS; START : in DAYS) is

    TODAY : DAYS := START;
    THE_COL : array (DAYS) of TEXT_IO.COUNT := (1, 7, 13, 19, 25, 31, 37);

    package INT_IO is new TEXT_IO.INTEGER_IO (NATURAL);
    package MONTH_IO is new TEXT_IO.ENUMERATION_IO (MONTHS);

begin
    TEXT_IO.NEW_LINE;

    MONTH_IO.PUT(MONTH);
    TEXT_IO.SET_COL (35);
    INT_IO.PUT (YEAR,4);
    TEXT_IO.NEW_LINE;

    TEXT_IO.PUT_LINE (" S    M    T    W    T    F    S");
    TEXT_IO.NEW_LINE;

    for THE_DAY in 1 .. LAST_DAY (YEAR, MONTH)
    loop
        TEXT_IO.SET_COL (THE_COL (TODAY));
        INT_IO.PUT (THE_DAY, 2);

        if TODAY = DAYS'LAST then
            TEXT_IO.NEW_LINE;
            TODAY := DAYS'FIRST
        else
            TODAY := DAYS'SUCC (TODAY);
        end if;
    end loop;

    TEXT_IO.NEW_LINE;

end PRINT_MONTH;
```


ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

```
with TEXT_IO;
package body ROMAN is

    type CONVERT_ARRAY is array (DIGIT) of NATURAL;

    DIGIT_TO_NATURAL : constant CONVERT_ARRAY
                                := (1, 5, 10, 50, 100, 500, 1000);

    procedure GET_VALID (RN : out NUMERAL) is separate;

    procedure PUT (RN : in NUMERAL) is separate;

    function CREATE (S : DIGIT_STRING) return NUMERAL is separate;

    function CONVERT (VN : VALID_NUMBER) return NUMERAL
                                                is separate;

    -- The preceding subprograms were represented as body stubs.
    -- Their associated subunits will be found on subsequent pages.
    -- The remaining subprograms must be represented as proper
    -- bodies because of the following rules:
    --
    --   1. The designators of all compilation units must be
    --   identifiers (operator symbols are not allowed).
    --
    --   2. The simple names of all subunits that have the same
    --   ancestor library unit must be distinct identifiers.
```

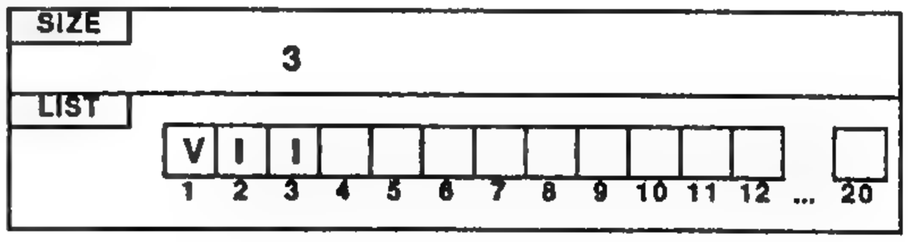
DIGIT_TO_NATURAL

'I'	1
'V'	5
'X'	10
'L'	50
'C'	100
'D'	500
'M'	1000

ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

```
    function "+" (LEFT, RIGHT : NUMERAL) return NUMERAL is
    begin
        return CONVERT (CONVERT (LEFT) + CONVERT (RIGHT ) );
    exception
        when CONSTRAINT_ERROR =>
            raise ILLEGAL_ROMAN_NUMERAL;
    end "+";


    function "<" (LEFT, RIGHT : NUMERAL) return BOOLEAN is
    begin
        return CONVERT (LEFT) < CONVERT ( RIGHT);
    end "<";


    function CONVERT (RN : NUMERAL) return VALID_NUMBER is

        SUM : NATURAL := 0;

    begin
        for INDEX in 1 .. RN.SIZE
        loop
            SUM := SUM + DIGIT_TO_NATURAL (RN.LIST (INDEX));
        end loop;

        return VALID_NUMBER (SUM);
    end CONVERT;
end ROMAN;
```

RN

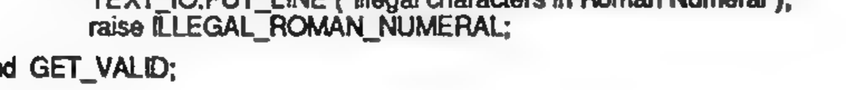


ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

```
separate (ROMAN)
procedure GET_VALID (RN : out NUMERAL) is
      STR     : STRING (1 .. 20);          -- The input string
      COUNT   : NATURAL;                   -- # of characters entered
      NUM     : DIGIT_STRING (1 .. 20);    -- Result of conversion
begin
      TEXT_IO.PUT_LINE ("Enter a roman numeral");
      TEXT_IO.GET_LINE (STR, COUNT);

      for CH in 1 .. COUNT
      loop
         NUM(CH) := DIGIT'VALUE(CHARACTER'IMAGE (STR (CH)));
      end loop;

      RN := CREATE (NUM (1 .. COUNT));  -- Pass DIGIT_STRING to the
                                        -- CREATE function.
exception
      when ILLEGAL_ROMAN_NUMERAL =>
          raise;          -- Error message was already printed in CREATE

      when CONSTRAINT_ERROR =>
          TEXT_IO.PUT_LINE ("Illegal characters in Roman Numeral");
          raise ILLEGAL_ROMAN_NUMERAL;
end GET_VALID;
```

COUNT

STR

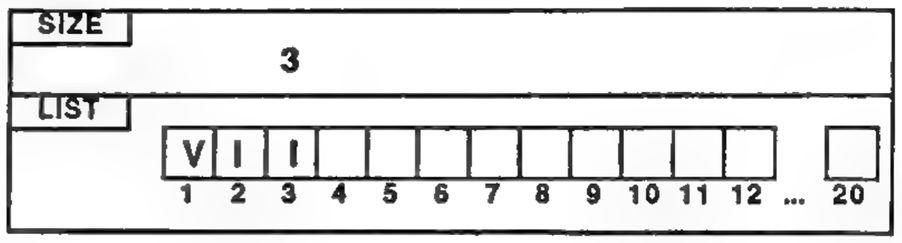


NUM

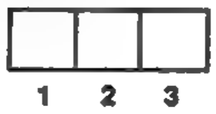
ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

```
separate (ROMAN)
procedure PUT (RN : in NUMERAL) is

    STR : STRING (1 .. RN.SIZE);

begin

    for CH in 1 .. RN.SIZE
    loop
        STR (CH) := CHARACTER'VALUE (DIGIT'IMAGE (RN.LIST (CH)));
    end loop;

    TEXT_IO.PUT (STR);

end;
```

RN



STR



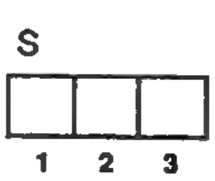
ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

```
separate (ROMAN)
function CREATE (S : DIGIT_STRING) return NUMERAL is
      RESULT : NUMERAL;
      LIMITS : constant CONVERT_ARRAY := (4, 1, 4, 1, 4, 1, 4);
      TOTAL : CONVERT_ARRAY := ('I' .. 'M' => 0);
begin
      -- Treat the first DIGIT separately
      TOTAL (S(1)) := TOTAL (S(1)) + 1;

      -- Check for out-of-order errors, sum up number of DIGITs
      for INDEX in 2 .. S'LENGTH
      loop
            TOTAL (S (INDEX)) := TOTAL (S (INDEX)) + 1;
            if S(INDEX) > S (INDEX - 1) then
                  TEXT_IO.PUT_LINE ("Digits out-of-order");
                  raise ILLEGAL_ROMAN_NUMERAL;
            end if;
      end loop;

      -- Check for correct number of each DIGIT
      for INDEX in DIGIT
      loop
            if TOTAL (INDEX) > LIMITS (INDEX) then
                  TEXT_IO.PUT_LINE ("Too many of a given digit");
                  raise ILLEGAL_ROMAN_NUMERAL;
            end if;
      end loop;

      -- S represents a valid NUMERAL
      RESULT.SIZE := S'LENGTH;
      RESULT.LIST (1 .. RESULT.SIZE) := S;

      return RESULT;

end CREATE;
```

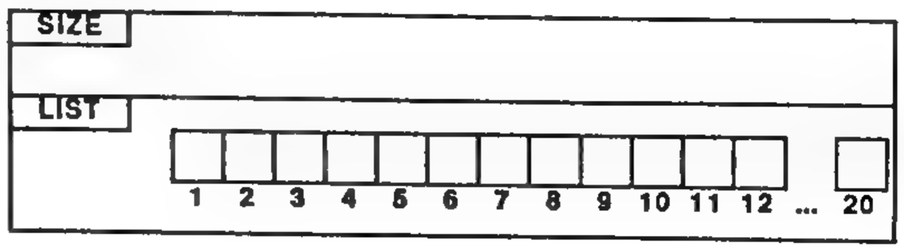

ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

	LIMITS		TOTAL
'I'	4	'I'	0
'V'	1	'V'	0
'X'	4	'X'	0
'L'	1	'L'	0
'C'	4	'C'	0
'D'	1	'D'	0
'M'	4	'M'	0

S



RESULT

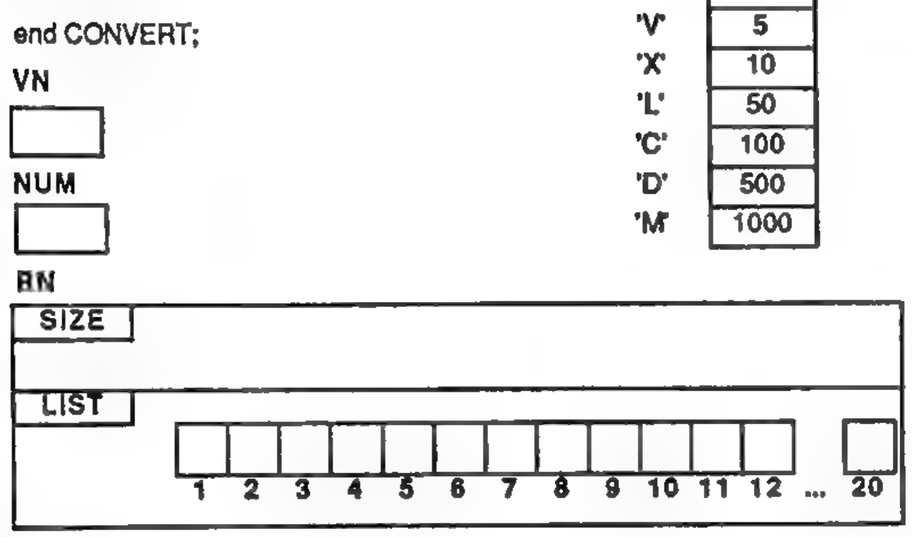


ROMAN NUMERAL PROBLEM (EXERCISE PG 7)

```
separate (ROMAN)
function CONVERT (VN : VALID_NUMBER) return NUMERAL is
      RN     : NUMERAL;
      NUM    : NATURAL := NATURAL (VN);
begin
      RN.SIZE := 0;

      for INDEX in reverse DIGIT  -- Try all DIGITs (beginning with 'M')
      loop
          -- spin through all occurences (if any) of this digit
          loop
              exit when DIGIT_TO_NATURAL (INDEX) > NUM;
              RN.SIZE := RN.SIZE + 1;
              RN.LIST (RN.SIZE) := INDEX;
              NUM := NUM - DIGIT_TO_NATURAL (INDEX);
          end loop;

      end loop;

      return RN;
end CONVERT;
```

DIGIT_TO_NATURAL

'I'	1
'V'	5
'X'	10
'L'	50
'C'	100
'D'	500
'M'	1000



CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
with TEXT_IO;
package body CHANGE_INFO is
   package MONEY_IO is new TEXT_IO.FLOAT_IO (MONEY_TYPE);

   package INT_IO is new TEXT_IO.INTEGER_IO (NATURAL);

   package DENOM_IO is new TEXT_IO.ENUMERATION_IO
                                          (CURRENCY_NAMES);

   procedure GET_INPUT ( PRICE: out MONEY_TYPE;
                         PAID  : out MONEY_TYPE) is separate;

            -- Initial 'stub':   PRICE := 2.37;
            --                   PAID := 20.00;

   function CHANGE_DUE ( PRICE : MONEY_TYPE;
                         PAID  : MONEY_TYPE)
                         return CANONICAL_UNITS is separate;

            -- initial 'stub':   return 1763;

   function MAKE CHANGE ( UNITS : CANONICAL_UNITS)
                                  return CURRENCY_LIST is separate;

                -- Initial 'stub':   return (3, 0, 1, 0, 1, 2, 1, 1);

    procedure PRINT_CURRENCY (MONEY : in CURRENCY_LIST)
                                                  is separate;


    function USER_WANTS_TO_STOP return BOOLEAN
                                                  is separate;

                -- Initial 'stub':   return TRUE;

end CHANGE_INFO;
```


CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
with CHANGE_INFO;
procedure CHANGE_MAKER is
    -- Set up actual generic parameters

    type DENOM is ( PENNY, NICKEL, DIME, QUARTER, HALF,
                    ONE, FIVE, TEN, TWENTY);

    type DENOM_LIST is array (DENOM) of NATURAL;
    MY_VALUES : constant DENOM_LIST :=
                (1, 5, 10, 25, 50, 100, 500, 1000, 2000);

    -- Create an instance of the generic package

    package U_S_CHANGE is new CHANGE_INFO
                (CURRENCY_NAMES   => DENOM,
                 CURRENCY_LIST    => DENOM_LIST,
                 CURRENCY_VALUES  => MY_VALUES);

    use U_S_CHANGE;

    -- Declare local objects to be used

    AMOUNT_CHARGED  : MONEY_TYPE;
    AMOUNT_PAID     : MONEY_TYPE;

begin
    loop
      GET_INPUT (AMOUNT_CHARGED, AMOUNT _PAID);

      PRINT_CURRENCY
          (MAKE_CHANGE
              (CHANGE_DUE (AMOUNT_CHARGED, AMOUNT_PAID)));

      exit when USER_WANTS_TO_STOP;

    end loop;

end CHANGE_MAKER;
```


CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
separate (CHANGE_INFO)
procedure GET_INPUT ( PRICE  : out MONEY_TYPE;
                      PAID   : out MONEY_TYPE) is

    PRICE_ENTERED     : MONEY_TYPE;
    PAYMENT_ENTERED : MONEY_TYPE;

    procedure INPUT (AMOUNT : out MONEY_TYPE) is separate;

begin
    TEXT_IO.PUT_LINE ("All values should have two decimal places");

    loop
        TEXT_IO.PUT ("PRICE: ");
        INPUT (PRICE_ENTERED);
        TEXT_IO.NEW_LINE;

        TEXT_IO.PUT ("PAID: ");
        INPUT (PAYMENT_ENTERED);
        TEXT_IO.NEW_LINE;

        exit when PAYMENT_ENTERED >= PRICE_ENTERED;
        TEXT_IO.PUT_LINE ("Insufficient payment; try again.");
    end loop;

    PRICE := PRICE_ENTERED;     -- send appropriate values
    PAID := PAYMENT_ENTERED;  -- back through the out parameters
end GET_INPUT;
```

CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
separate (CHANGE_INFO.GET_INPUT)
procedure INPUT (AMOUNT : out MONEY_TYPE) is

begin
      loop
            begin
                  MONEY_IO.GET(AMOUNT);
                  exit;
            exception
                  when TEXT_IO.DATA_ERROR =>
                        TEXT_IO.SKIP_LINE;
                        TEXT_IO.PUT_LINE (" ERROR : Input value again");
                  when CONSTRAINT_ERROR =>
                        TEXT_IO.PUT_LINE (" ERROR : Input value again");
            end;
      end loop;
end INPUT;
```

```
separate (CHANGE_INFO)
function CHANGE_DUE ( PRICE  : MONEY_TYPE;
                      PAID   : MONEY_TYPE)
                      return CANONICAL_UNITS is

begin

      return CANONICAL_UNITS ( ( PAID - PRICE ) * 100.0 );

end CHANGE_DUE;
```


CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
separate (CHANGE_INFO)
function MAKE_CHANGE (UNITS : CANONICAL_UNITS)
                      return CURRENCY_LIST is

      RESULT : CURRENCY_LIST;
      COINS  : CANONICAL_UNITS := UNITS;

begin
      for INDEX in CURRENCY_LIST'RANGE
      loop
        RESULT (INDEX) := COINS   /   CURRENCY_VALUES (INDEX);
        COINS          := COINS MOD CURRENCY_VALUES (INDEX);
      end loop;

      return RESULT;

end MAKE_CHANGE;
```

CURRENCY_VALUES	
PENNY	1
NICKEL	5
DIME	10
QUARTER	25
HALF	50
ONE	100
FIVE	500
TEN	1000
TWENTY	2000

RESULT	
PENNY	
NICKEL	
DIME	
QUARTER	
HALF	
ONE	
FIVE	
TEN	
TWENTY	

UNITS

COINS

CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
separate (CHANGE_INFO)
procedure PRINT_CURRENCY (MONEY : in CURRENCY_LIST) is

begin

      for INDEX in CURRENCY_LIST'RANGE
      loop
            if MONEY (INDEX) > 0 then

                  DENOM_IO.PUT (INDEX);
                  TEXT_IO.SET_COL(12);
                  TEXT_IO.PUT ("=");
                  INT_IO. PUT (MONEY (INDEX));
                  TEXT_IO.NEW_LINE;

            end if;

      end loop;

end PRINT_CURRENCY;
```


CHANGE MAKER PROBLEM (EXERCISE PG 9)

```
separate (CHANGE_INFO)
function USER_WANTS_TO_STOP return BOOLEAN is

      RESPONSE     : STRING (1 .. 10);
      COUNT        : NATURAL;
begin
      TEXT_IO.PUT_LINE ("Do you want to enter another pair" &
                                    " of amounts (Y or N) ");

      TEXT_IO.GET_LINE (RESPONSE, COUNT);

      return RESPONSE (1) = 'N' or RESPONSE (1) = 'n';

exception

      when others =>

            TEXT_IO.PUT_LINE ("Illegal input -- 'No' assumed");
            return TRUE;

end USER_WANTS_TO_STOP;
```

		CAT	DEP	RULE
1.	procedure PLAY_AdaVENTURE	LIB	2	1
2.	package COMMAND_INFO	LIB	4	1
3.	package body COMMAND_INFO	SEC	2	2
4.	package VOCABULARY	LIB	XXX	XXX
5.	procedure GET	SEC	3/21	3/1
6.	procedure EXECUTE	SEC	3/21	3/1
7.	function USER_QUITS	SEC	3	3
8.	function USER_WINS	SEC	3	3
9.	procedure GO_RTN	SEC	6/14	3/1
10.	package PLAYER	LIB	4	1
11.	package body PLAYER	SEC	10	2
12.	package DUNGEON_DOOR	LIB	XXX	XXX
13.	package body DUNGEON_DOOR	SEC	12	2
14.	package MAP	LIB	4	1
15.	package body MAP	SEC	12/14/21	1/2/1
16.	procedure LIST_INVENTORY	SEC	11/21	3/1
17.	function POSITION_OF_BLANK	SEC	5	3
18.	function STRING_TO_WORDS	SEC	5	3
19.	function TRANSFORM_1	SEC	5	3
20.	function TRANSFORM_2	SEC	5	3
21.	package TEXT_IO	LIB	XXX	XXX